Politecnico di Torino
Universitat Politècnica de Catalunya, BarcelonaTech

MASTER OF SCIENCE IN COMPUTER ENGINEERING

Final Project

Integrated Policy Management Framework for the EMOTIVE Cloud Middleware

EMOTIVE Cloud: the Barcelona Supercomputing Center's IaaS open-source solution for Cloud Computing

Supervisors:
Prof. Fulvio Risso, *Politecnico di Torino*
Dr. Jordi Guitart, *UPC*
Ing. Guido Marchetto, *Politecnico di Torino*

Mauro Canuto

September 2012

Acknowledgements

This research project would not have been possible without the support of many people. I would like to express my gratitude to Dr. Jordi Guitart for being an outstanding advisor and excellent professor whose expertise, understanding, and patience, added considerably to my graduate experience. Deepest gratitude are also due to the other members of the research group, especially Josep Subirats, for the assistance and technical support he provided at all levels of the project. Also, I cannot but express my sincere gratitude to Prof. Fulvio Risso who has been my first significant figure in Italy during my studies abroad. His thoughtful advice often served to give me a sense of direction during this year.

I would wishes to express my love and gratitude to my family and my sister for the support they provided me through my entire life. I am tempted to individually thank all of my friends and all the people I knew in Barcelona during my Erasmus but as the list might be long and for fear I might omit someone, I will simply and genuinely say: Thank you to you all for your love, care and trust. In particular I would like to cite: Marco, Matteo, Enrico, Clemens, Federico and Cesare for their constant presence and care and instant moral support.

Without all of them, everything would be more difficult.

Mauro Canuto

Contents

List of Figures

Glossary

Virtual Machine (VM): software implementation of a computing environment.

Paravirtualization: virtualization technique that allows the operating system to be aware that it is running on a hypervisor instead of base hardware. The operating system must be modified to accommodate the unique situation of running on a hypervisor instead of basic hardware.

Hardware Virtual Machine (HVM): operating system that is running in a virtualized environment unchanged and unaware that it is not running directly on the hardware. Special hardware is required to allow this.

Service Level Agreement (SLA): contractual agreement by which a service provider defines the level of service, responsibilities, priorities, and guarantees regarding availability, performance, and other aspects of the service.

Quality of Service (QoS): combination of different complementary technologies that combine to provide different users and different applications with the specific network performance guarantees that allow them to get their jobs done efficiently and quickly.

High-Performance Computing (HPC): a term that refers to the use of parallel processing for running advanced application programs efficiently, reliably and quickly.

Amazon machine image (AMI): encrypted machine image stored in Amazon S3. It contains all the information necessary to boot instances of your software.

Open Virtualization Format (OVF): specification that describes an open, secure, portable, efficient and extensible format for the packaging and distribution of software to be run in virtual machines. http://dmtf.org/standards/ovf

Common Information Model (CIM): open standard that defines how managed elements in an IT environment are represented as a common set of objects and relationships between them.

Abstract Syntax Tree (AST): tree representation of the abstract syntactic structure of source code written in a programming language.

Extended Backus-Naur Form (EBNF): family of metasyntax notations used for expressing context-free grammars (EBNF standard : ISO/IEC 14977).

1

Introduction

1.1 Motivation

Cloud computing is being more and more important in IT infrastructure because of its new levels of efficiency, flexibility and cost saving, particularly in the area of the infrastructure-as-a-service (IaaS). It has achieved to move centralized physical resources to shared virtual resources, reducing costs and maintenance because we can accommodate more than one virtual machine in the same host, avoiding the expense traditional physical machine for only one service. Virtualization plays an important role in this opening a wide new range of opportunities for managing computer resources mostly because it allows isolating different tasks in a single node without any interference with others. Moreover, a virtual machine is easier to monitor and offers a powerful but simple interface for managing its allocated resources which brings new chances for resource management. In addition to this, machine maintenance acquires a new level thanks to virtualization since it allows managing machines as software making it a perfect environment for executing tasks and complex services.

1.2 Goals

At UPC (Universitat Politècnica de Catalunya, BarcelonaTech) and BSC (Barcelona Supercomputing Center) a Cloud platform based on virtualizated environments has been developed: EMOTIVE Cloud (Elastic Management Of Tasks In Virtualized Environments - *www.emotivecloud.net*). EMOTIVE main feature is VMs management with various scheduling policies and its framework has multiple schedulers with different

capabilities such as machine learning, prediction, economic, fault tolerance, semantic description, or SLA enforcement. In this sense, it can use a simplistic Round Robin, or a consolidation aware scheduling like Backfilling but in the current system the policy has to be pre-defined in the code and it cannot be changed run-time.

This project main aim is to expand and evolve the capabilities of EMOTIVE platform to improve certain limitations such as new scheduling management and policy definitions implementing a policy engine within the EMOTIVE middleware that can load and enforce dynamically policies for virtual machines placement and scheduling operations in this Cloud infrastructure. This new framework will allow user to load and use scheduling policies that are not fixed in the code but that can be modified depending on the needs and implemented without recompiling the whole sources.

This has been achieved thanks to the usage of a common policy language and interface which allows developing new schedulers with different features and policies and load them through RESTful Web Services. It has been necessary to develop a compiler able to accept policy statements and translate them into sequences of machine language operations which, loaded into memory and executed, carry out the intended computation. After that the compiler has been integrated in the EMOTIVE system so when a user decides to change the running scheduler it will parse, execute and load the new policy without the need to re-compile the whole code.

The main tasks needed to be done can be summarized as follows:

1. *Familiarization with IaaS solutions for Cloud Computing*: before starting to work on this project, in order to understand better the environments and to fix some gaps, I had to deepen my knowledge in Cloud Computing, especially related to IaaS (Infrastructure-as-a-Service). For this reason I analyzed and examined toolkit for managing distributed infrastructures such as OpenNebula, OpenStack, Amazon EC2 and Eucalyptus.

2. *Software installation and familiarization with EMOTIVE Cloud*: EMOTIVE requires the installation and the setup of base software in the nodes that will be part of the systems. At first I installed all the environments I needed for running EMOTIVE, such as Xen (the hypervisor whose layer supports one or more guest

operating systems) and Libvirt (a toolkit to interact with the virtualization capabilities of recent versions of Linux) in order to deepen their features then I setted up a VPN network to test the system in a cloud with 2 nodes.

3. *EMOTIVE source code analysis*: in order to become familiar with the EMOTIVE source code, I analyzed its structure and its modules, especially the DRP and SimpleScheduler which are responsible of loading the system policies and the Virtualized Resource Management and Monitoring (VRMM) whose first capability is the task of creating and maintaining virtual machines, which is done by the Virtualization Manager (VtM).

4. *Study of languages to define policies and tools to parse them*: the first strategy to follow in order to achieve the project goals was to model a policy using an XML W3C schema and trying to load it dynamically using the JAXB APIs. Following this strategy I noticed that it would have been very complex to define different policies with a more specific syntax. Hence I decided to develop a new parser from scratch in order to define policies taking advantage of an higher flexibility due to the lexical analysis and of a compiler done for this project purposes. The analysis of other policy specification languages, such as Ponder (section 2.3.1), helped me to understand the path to follow in order to develop this framework.

5. *Definition of LEPIC language*: before defining the language to implement I examined in detail which features policies need to have and which EMOTIVE APIs can be useful in scheduling operations. Then I defined the grammar and the productions in order to provide a semantic and lexical analysis that a policy definition must comply with.

6. *Implementation of tools for parsing and code generation with LEPIC*: after having evaluated different parser generator such as JavaCC and Bison I ended up choosing SableCC which is a bottom up parser that takes an unconventional and interesting approach of using object oriented methodology for constructing parsers. It also keeps a clean separation between machine-generated code and user-written code which leads to a shorter development cycle.

7. *Integration and implementation of policy management framework in EMOTIVE supporting LEPIC*: this is the main task that makes the new framework working

with the EMOTIVE middleware. This goal has been achieved implementing new classes and making some changes in the scheduling layer. Moreover some functions have been developed so as LEPIC can interact with EMOTIVE, taking advantage of some of its features.

8. *Case studies to test the policy framework*: after having implemented the new framework we tested it by running different policies and evaluating performance. During this part we also found out some tasks than could be done in a future work.

9. *Documentation of the project*: the last part of this project was devoted to write the documentation about the LEPIC language features and the new implemented features of EMOTIVE.

1.3 Project planning

The first part of project management was to plan and subsequently report progress within the project environment in order to have a planning in which the project scope is defined and the appropriate methods for completing the project are determined. Following this step, the durations for the various tasks necessary to complete the work were listed and costs, in terms of time, for each activity was estimated. The diagram 1.1 shows the original planning and the evaluated time considered necessary to achieve the objectives, compared with the required time.

Small deviations from the original plan are mainly due to the need to deepen the analysis of EMOTIVE middleware, especially some parts of its source code, or to solve unexpected behaviours of the system.

1.4 Document structure

This document begins with an introduction about some backgrounds and related work, including an analysis of other projects concerning virtualized data centres and cloud infrastructures.

Chapter 3 introduces the technical aspects of EMOTIVE, describing its structure and implementation, especially the scheduling part. That section is followed by the presentation of the LEPIC language and the creation of policies using particular rules.

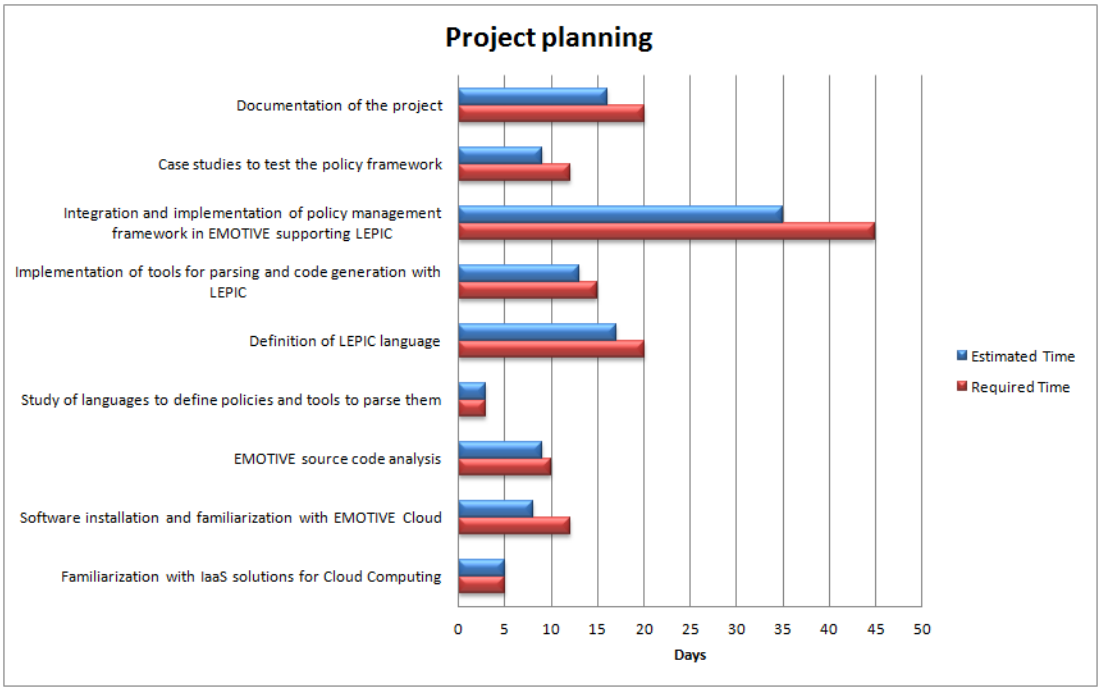


Figure 1.1: Project planning

Then the integration of the policy framework in EMOTIVE is described and followed by cases of study and tests using this framework, by referring to policies created with the LEPIC language, proposing also possible future work related to this project.

2

Background and related work

2.1 Virtualization

Before taking a deeper dive into into the practical aspects of this project it is useful to introduce virtualization concepts and technologies and Cloud Computing infrastructures:

- Virtualization: "The ability to run multiple operating systems on a single physical system and share the underlying hardware resources" (*VMware white paper, Virtualization Overview*)
- Cloud Computing: "The provisioning of services in a timely (near on instant), on-demand manner, to allow the scaling up and down of resources" (*Alan Williamson, quoted in Cloud BootCamp March 2009*)

Starting from these definitions we are going to describe what Cloud Computing and Virtualization can offer and services available nowadays.

Virtual servers seek to encapsulate the server software away from the hardware. This includes the OS, the applications, and the storage for that server. Servers end up as mere files stored on a physical box, or in enterprise storage and a virtual server can be serviced by one or more hosts, and one host may house more than one virtual server.

In order to manage virtual machines in a virtualized environment the use of an hypervisor is necessary. A hypervisor also known as a virtual machine manager/monitor (VMM), is a computer hardware platform virtualization software that allows several operating systems to share a single hardware host. Each operating system appears to

have the host's processor, memory, and resources to itself. Instead, the hypervisor is controlling the host processor and resources, distributing what is needed to each operating system in turn and ensuring that the guest operating systems/virtual machines are unable to disrupt each other.

Figure 2.1: Host with hypervisor

Although there are many products that offer virtualization technologies and that are supported by EMOTIVE, such as KVM (Kernel-based Virtual Machine with a Linux kernel virtualization infrastructure) and Virtual Box, for this project we worked mainly using a Xen environment.

2.1.1 Xen hypervisor

The Xen hypervisor is a layer of software running directly on computer hardware replacing the operating system thereby allowing the computer hardware to run multiple guest operating systems concurrently. Support for x86, x86-64, Itanium, Power PC, and ARM processors allow Xen to run on a wide variety of computing devices and currently supports Linux, NetBSD, FreeBSD, Solaris, Windows, and other common operating systems as guests running on the hypervisor.

By separating the guests from the hardware, the Xen hypervisor is able to run multiple operating systems securely and independently and it runs directly on the hardware becoming the interface for all hardware requests such as CPU, I/O, and disk for the guest operating systems.

The Domain 0 Guest referred to as Dom0 is launched by the Xen hypervisor during initial system start-up and can run any operating system except Windows. The Dom0 has unique privileges, allow it to manage all aspects of Domain Guests such as starting,

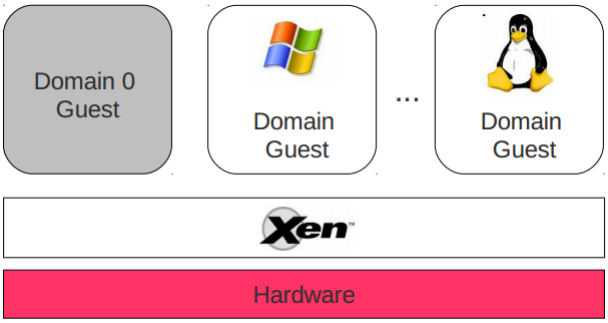


Figure 2.2: Components of a computer running Xen

stopping, I/O requests, etc., to access the Xen hypervisor that is not allocated to any other Domain Guests. The Domain Guests referred to as DomUs are launched and controlled by the Dom0 and independently operate on the system. These guests are either run with a special modified operating system referred to as paravirtualizion or un-modified operating systems leveraging special virtualization hardware (Intel VT and AMD-V) referred to as hardware virtual machine (HVM).

For the Xen server to function properly, the *xend* daemon must be running on the VM server. This daemon must be running to start and manage the virtual machines. The administrative interface of xend is "xm". The xend daemon can be customized using the configuration file /etc/xen/xend-config.sxp. This daemon allows the administrators to access the hypervisors and the virtual machines using management tools like "virsh" and "virt-install". In the following lines there are some xm and virsh commands for creating, managing, and troubleshooting xen virtual machines.

- **xm list**: this command displays the information of all running domains.

```
pctinet:~# xm list
Name                                        ID   Mem VCPUs      State   Time(s)
Domain-0                                     0 14618     4     r-----  78650.0
vm1                                         82   256     4     -b----      3.7
vm2                                         83   256     4     -b----      3.6
vm3                                         85   256     4     -b----      1.9
```

Figure 2.3: xm list command

  - Name - represents the domU guest VM name
  - ID -represents the domain ID

  - Mem - represents the amount of memory allotted to the guest domain (in MB)
  - VCPUs - represents the number of virtual CPUs assigned to a domain.
  - State - represents the running state of the guest OS.
    * r - running: The domain is currently running on a CPU
    * b - blocked : The domain is blocked, and not running or runnable. This can be caused because the domain is waiting on IO (a traditional wait state) or has gone to sleep because there was nothing else for it to do.
    * p - paused : The domain has been paused, usually occurring through the administrator running xm pause. When in a paused state the domain will still consume allocated resources like memory, but will not be eligible for scheduling by the Xen hypervisor.
    * s - shutdown : The guest has requested to be shutdown, rebooted or suspended, and the domain is in the process of being destroyed in response.
    * c - crashed : The domain has crashed, which is always a violent ending. Usually this state can only occur if the domain has been configured not to restart on crash. See xmdomain.cfg for more info.
    * d - dying : The domain is in process of dying, but hasn't completely shutdown or crashed.

- **xm create**: This is used to startup a Xen guest. This command creates a guest based on the configuration file storage in /etc/xen.
- **xm destroy** $< domain - id >$: This command is used to immediately terminate an Xen domU guest virtual machine.
- **xm console** $< domain - id >$:Attach to domain domain-id's console. If you have set up your Domains to have a traditional log in console this will look much like a normal text log in screen.

2.1.2 Libvirt

Libvirt is a virtualization API and a daemon for managing virtual machines (VMs), remote or locally, using multiple virtualization back-ends. It allows to manage different

Virtualization solutions such as KVM and Xen through a common (programming and user) interface.

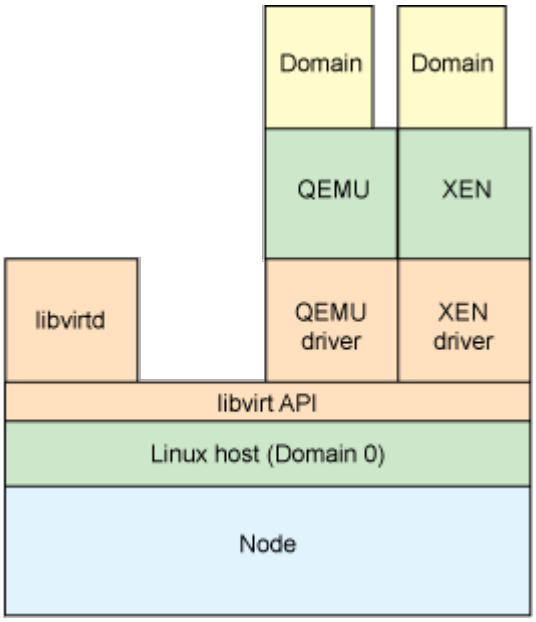


Figure 2.4: Driver-based architecture of libvirt

When running in a Xen environment, programs using libvirt have to execute in "Domain 0" and the Xen daemon supervises the control and execution of the sets of domains. The hypervisor, drivers, kernels and daemons communicate though a shared system bus implemented in the hypervisor. The library usually interacts with the Xen daemon for any operation changing the state of the system, but for performance and accuracy reasons may talk directly to the hypervisor when gathering state information at least when possible (i.e. when the running program using libvirt has root privilege access).

2.1.2.1 Libvirt API and virsh commands

This project has been developed using JAVA language, and the following table lists shows Libvirt API main methods that have been involved:

initialize	destroy	getNodeCPUNum
getDomain	pause / unpause	getNodeCPUSpeed
getDomainID	save	getCPUNum
getDomainNameList	restore	setCPU
create	migrate	updateCache

The virsh program is the main interface for managing virsh guest domains. The program can be used to create, pause, and shutdown domains. It can also be used to list current domains. Here there are some options that the virsh program understands:

- *list optional –inactive –all* : prints information about one or more domains. If no domains are specified it prints out information about running domains. An example format for the list is as follows:

Id	Name	State
0	Domain-0	running
2	fedora	paused

  Name is the name of the domain. ID the domain numeric id. State is the run state (see below).

- *create FILE optional –console –paused*: create a domain from an XML file. An easy way to create the XML file is to use the dumpxml command to obtain the definition of a pre-existing guest. The domain will be paused if the –paused option is used and supported by the driver; otherwise it will be running. If –console is requested, attach to the console after creation.

- *console domain-id [devname]*: connect the virtual serial console for the guest. The optional devname parameter refers to the device alias of an alternate console, serial or parallel device configured for the guest. If omitted, the primary console will be opened.

- *destroy domain-id*: immediately terminate the domain domain-id. This doesn't give the domain OS any chance to react, and it's the equivalent of ripping the power cord out on a physical machine.

- *help optional command-or-group*: this lists each of the virsh commands. When used without options, all commands are listed, one per line, grouped into related categories, displaying the keyword for each group. To display only commands for a specific group, give the keyword for that group as an option.

2.2 Cloud computing

Cloud computing has emerged as a new computing paradigm which aims to provide reliable, customized and QoS guaranteed dynamic computing environments for end-users. With cloud computing, it is possible to enable more flexible service delivery and automate core IT processes, including both user and application provisioning and systems management. These services are broadly divided into three categories: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS).

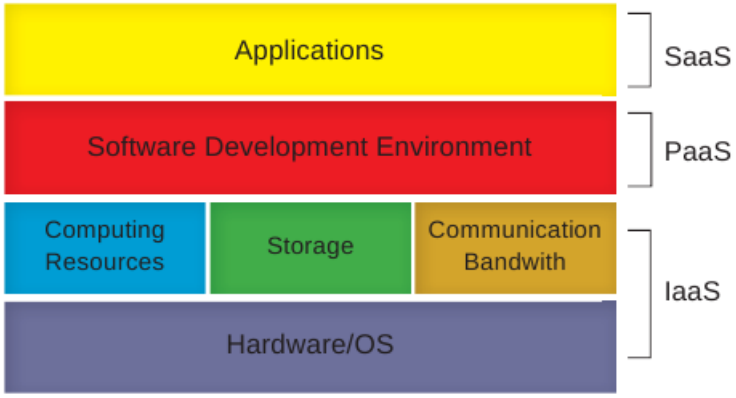


Figure 2.5: Cloud layers

- *Infrastructure-as-a-Service (IaaS)*: it provides grids, clusters or virtualized servers, networks, storage and systems software designed to augment or replace the functions of an entire data center.

- *Platform-as-a-Service (PaaS)*: it provides virtualized servers on which users can run existing applications or develop new ones without having to worry about maintaining the operating systems, server hardware, load balancing or computing capacity.

- *Software-as-a-Service (SaaS)*: this is the most widely known and widely used form of cloud computing. SaaS provides all the functions of a sophisticated traditional application, but through a Web browser, not a locally-installed application. It eliminates worries about application servers, storage, application development and related, common concerns of IT.

Cloud computing comes in three forms: public clouds, private clouds, and hybrids clouds (Figure 2.6).

- **Public Clouds** : A public cloud is one in which the services and infrastructure are provided off-site over the Internet. These clouds offer the greatest level of efficiency in shared resources; however, they are also more vulnerable than private clouds. Many IT department executives are concerned about public cloud security and reliability. Take extra time to ensure that you have security and governance issues well planned, or the short-term cost savings could turn into a long-term nightmare.

- **Private Clouds** : A private cloud is one in which the services and infrastructure are maintained on a private network. These clouds offer the greatest level of security and control, but they require the company to still purchase and maintain all the software and infrastructure, which reduces the cost savings.

- **Hybrid Clouds** : A hybrid cloud includes a variety of public and private options with multiple providers. By spreading things out over a hybrid cloud, you keep each aspect at your business in the most efficient environment possible. The downside is that you have to keep track of multiple different security platforms and ensure that all aspects of your business can communicate with each other.

The next section describes some Cloud Middleware whose functionalities are comparable to EMOTIVE, with particular attention to the scheduling layer.

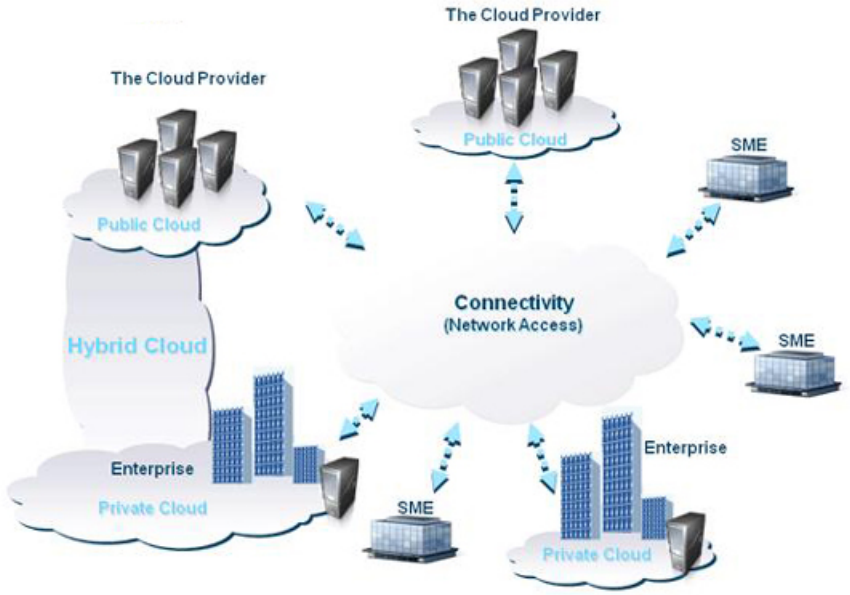


Figure 2.6: Public, private and hybrid clouds

2.2.1 OpenNebula

OpenNebula is an open-source cloud computing toolkit for managing heterogeneous distributed data center infrastructures. It can be primarily used as a virtualization tool to manage virtualized infrastructure in the data center or cluster, which is usually referred as Private Cloud. OpenNebula supports Hybrid Cloud to combine local infrastructure with public cloud-based infrastructure, enabling highly scalable hosting environments and Public Clouds by providing Cloud interfaces to expose its functionality for virtual machine, storage and network management.

The toolkit includes features for integration, management, scalability, security and accounting. It also emphasizes standardization, interoperability and portability, providing cloud users and administrators with a choice of several cloud interfaces (EC2 Query, OGF OCCI and vCloud) and hypervisors (Xen, KVM and VMware), and a flexible architecture that can accommodate multiple hardware and software combinations in a data center.

2.2.1.1 Scheduler

The OpenNebula scheduling framework is designed in a generic way, so it is modifiable and can be easily replaced by third-party developments. OpenNebula comes with a match making scheduler (*mm_sched*) that implements the Rank Scheduling Policy. The goal of this policy is to prioritize those resources more suitable for the VM. The behaviour of the scheduler can be tuned to adapt it to your infrastructure with the following configuration parameters defined in /etc/oned/sched.conf:

- **ONED_PORT**: Port to connect to the OpenNebula daemon oned (Default: 2633)
- **SCHED_INTERVAL**: Seconds between two scheduling actions (Default: 30)
- **MAX_VM**: Maximum number of Virtual Machines scheduled in each scheduling action (Default: 300)
- **MAX_DISPATCH**: Maximum number of Virtual Machines actually dispatched to a host in each scheduling action (Default: 30)
- **MAX_HOST**: Maximum number of Virtual Machines dispatched to a given host in each scheduling action (Default: 1)
- **DEFAULT_SCHED**: Definition of the default scheduling algorithm

This scheduler algorithm easily allows the implementation of several placement heuristics depending of the RANK expression used:

Packing Policy (RANK = RUNNING_VMS)

- *Target*: Minimize the number of cluster nodes in use
- *Heuristic*: Pack the VMs in the cluster nodes to reduce VM fragmentation
- *Implementation*: Use those nodes with more VMs running first

Striping Policy (RANK = "- RUNNING_VMS")

- *Target*: Maximize the resources available to VMs in a node
- *Heuristic*: Spread the VMs in the cluster nodes

- *Implementation*: Use those nodes with less VMs running first

Load-aware Policy (RANK = FREECPU)

- *Target*: Maximize the resources available to VMs in a node
- *Heuristic*: Use those nodes with less load
- *Implementation*: Use those nodes with more FREECPU first

Although it is possible to select the preferred policy it is not possible to create new policies and load them while the system is running. This is the new feature that EMOTIVE provide to the user thanks to the LEPIC language and the integrated policy management framework.

2.2.2 OpenStack

OpenStack is an Infrastructure as a Service (IaaS) cloud computing project by Rackspace Cloud and NASA. Currently more than 150 companies have joined the project among which are AMD, Intel, Canonical, SUSE Linux, Red Hat, Cisco, Dell, HP, IBM and Yahoo. It is free open source software released under the terms of the Apache License and its system controls large pools of compute, storage, and networking resources throughout a datacenter, all managed through a dashboard that gives administrators control while empowering their users to provision resources through a web interface.

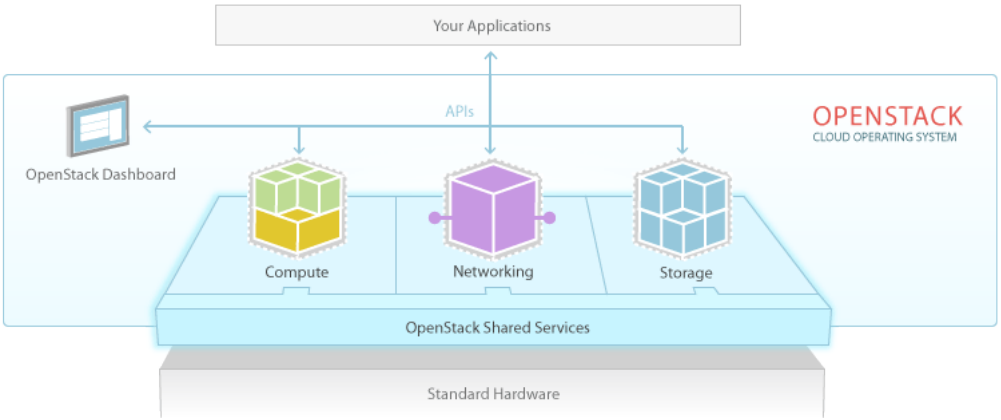


Figure 2.7: Openstack diagram

OpenStack is architected to provide flexibility as you design your cloud, with no proprietary hardware or software requirements and the ability to integrate with legacy

systems and third party technologies. It is designed to manage and automate pools of compute resources and can work with widely available virtualization technologies, as well as bare metal and high-performance computing (HPC) configurations.

Administrators often deploy OpenStack Compute using one of multiple supported hypervisors in a virtualized environment. KVM and XenServer are popular choices for hypervisor technology and recommended for most use cases. Linux container technology such as LXC is also supported for scenarios where users wish to minimize virtualization overhead and achieve greater efficiency and performance. In addition to different hypervisors, OpenStack supports ARM and alternative hardware architectures.

2.2.2.1 Scheduler

Openstack runs as a daemon named nova-schedule which picks up a compute server from a pool of available resources depending upon the scheduling algorithm in place. A scheduler can base its decisions on various factors such as load, memory, physical distance of the availability zone, CPU architecture, etc. and it implements a pluggable architecture. Currently the nova-scheduler implements a few basic scheduling algorithms:

- *Chance*: In this method, a compute host is chosen randomly across availability zones.
- *Availability zone*: Similar to chance, but the compute host is chosen randomly from within a specified availability zone.
- *Simple*: In this method, hosts whose load is least are chosen to run the instance. The load information may be fetched from a load balancer.

Compute is configured with the following default scheduler options:

```
scheduler_driver=nova.scheduler.multi.MultiScheduler
volume_scheduler_driver=nova.scheduler.chance.ChanceScheduler
compute_scheduler_driver=nova.scheduler.filter_scheduler.FilterScheduler
scheduler_available_filters=nova.scheduler.filters.standard_filters
scheduler_default_filters=AvailabilityZoneFilter,RamFilter,ComputeFilter
least_cost_functions=nova.scheduler.least_cost.compute_fill_first_cost_fn
compute_fill_first_cost_fn_weight=-1.0
```

Compute is configured by default to use the Multi Scheduler, which allows the admin to specify different scheduling behaviour for compute requests versus volume requests.

The volume scheduler is configured by default as a Chance Scheduler, which picks a host at random that has the nova-volume service running.

In the default configuration, this scheduler will only consider hosts that are in the requested availability zone (*AvailabilityZoneFilter*), that have sufficient RAM available (*RamFilter*), and that are actually capable of servicing the request (*ComputeFilter*).

From the resulting filtered list of eligible hosts, the scheduler will assign a cost to each host based on the amount of free RAM (*nova.scheduler.least_cost.compute_fill_first_cost_fn*), will multiply each cost value by -1 (*compute_fill_first_cost_fn_weight*), and will select the host with the minimum cost. This is equivalent to selecting the host with the maximum amount of RAM available.

2.2.3 Amazon Elastic Compute Cloud (EC2)

Amazon Elastic Compute Cloud (EC2) is a central part of Amazon.com's cloud computing platform, Amazon Web Services (AWS). EC2 allows users to rent virtual computers on which to run their own computer applications.

It is designed to make web-scale computing easier for developers and its simple web service interface allows to obtain and configure capacity with minimal friction. It provides a complete control of computing resources and lets run on AmazonâĂŹs proven computing environment.

In order to use Amazon EC2 is necessary to select a pre-configured, templated Amazon Machine Image (AMI) to get up and running immediately or create an AMI containing your applications, libraries, data, and associated configuration settings. Once an instance type(s) has been chosen it is possible to start, terminate, and monitor as many instances of AMI as needed, using the web service APIs or the variety of management tools provided. User pays only for the resources that he actually consumes, like instance-hours or data transfer.

2.2.4 Eucalyptus

This software platform for on-premise (private) IaaS clouds uses existing infrastructure to create a scalable, secure web services layer that abstracts compute, network and storage. Eucalyptus takes advantage of modern infrastructure virtualization software

to create elastic pools that can be dynamically scaled up or down depending on application workloads. Eucalyptus web services are uniquely designed for hybrid clouds using the industry standard Amazon Web Services API. The benefits are highly efficient scalability, increased trust and control for IT as a Service.

2.2.4.1 Scheduling policies

In Eucalyptus administrator can set one of the following three scheduling policies:

- *Greedy*: first node that is found that can run the VM will be chosen.
- *Round Robin*: nodes are selected one after another until one is found that can run the VM.
- *Powersave*: nodes are put to sleep when they are not running VMs, and reawakened when new resources are required. VMs will be placed on the first awake machine, followed by machines that are asleep.

2.3 Policy definition and management

Policies are rules that govern the choices in behaviour of a system and their management defines what actions need to be carried out when specific events occur within a system or what resources must be allocated under specific conditions. There is considerable interest in the use of policies for the security and management of large-scale networks and distributed services however there has been very little work on how to disseminate policies to the entities that will interpret them and how to deal with dynamic large-scale environments where the set of objects to which policies apply change, and where the policies themselves need to be updated to cater for changing requirements.

In order to create a policy framework according to the EMOTIVE needs we examined an existing policy deployment model developed at the Imperial College of London: Ponder.

2.3.1 Ponder overview

Ponder is a declarative, object-oriented language for specifying security policies with role-based access control, as well as general-purpose management policies for specifying

what actions are carried out when specific events occur within the system or what resources to allocate under specific conditions. Unlike many other policy specification notations, Ponder supports typed policy specifications. Management Structures Policies can be written as parametrised types, and the types instantiated multiple times with different parameters in order to create new policies. Furthermore, new policy types can be derived from existing policy types, supporting policy extension through inheritance.

Ponder has four basic policy types: authorisations, obligations, refrains and delegations and three composite policy types: roles, relationships and management structures that are used to compose policies. The dependencies between the various types are shown in figure 2.8.

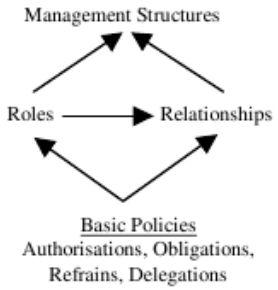


Figure 2.8: Ponder policy definitions

Ponder also has a number of supporting abstractions that are used to define policies: domains for hierarchically grouping managed objects, events for triggering obligation policies, and constraints for controlling the enforcement of policies at runtime.

Although EMOTIVE policies deal more with VMs placement an resources utilization than security aspects, Ponder analysis helped us to understand and to define some features which were not clear at the beginning (E.g. using parametrised types and providing the user with a schema to follow in order to create new scheduling operations).

3

EMOTIVE middleware

3.1 Introduction

This chapter introduces Elastic Management of Tasks in Virtualized Environments (EMOTIVE), the Barcelona Supercomputing Center (BSC)'s and BarcelonaTech (UPC) IaaS open-source solution for Cloud Computing. EMOTIVE provides users with elastic fully customized virtual environments in which to execute their applications. Further, it simplifies the development of new middleware services for managing Cloud systems by supporting resource allocation and monitoring, data management, live migration, and checkpoints. These features and its facility to be extended and configured make EMOTIVE especially appropriate to support research on Cloud Computing scenarios. Offering functionality comparable to its commercial counterparts allows EMOTIVE to be used on production to set up small Cloud platforms. It can be also used as a cloud provider and is very easy to extend thanks to its modular Web Services architecture. Furthermore, it also has the capability to use external resources, like from the public Cloud of Amazon EC2. This feature allows to be involved in a Cloud federation (insourcing/outsorcing) and create public, private and hybrids clouds.

3.2 Original Architecture

EMOTIVE Cloud is mainly composed by three different layers: the data infrastructure, the node management (VRMM), and the global Scheduler. The data infrastructure offers a distributed storage for supporting virtualization capabilities such as migration and checkpoint support. VRMM is in charge of creating and maintaining the whole

virtual machine lifecycle. Finally, the scheduling layer is in charge of distributing tasks and VMs among the physical nodes.

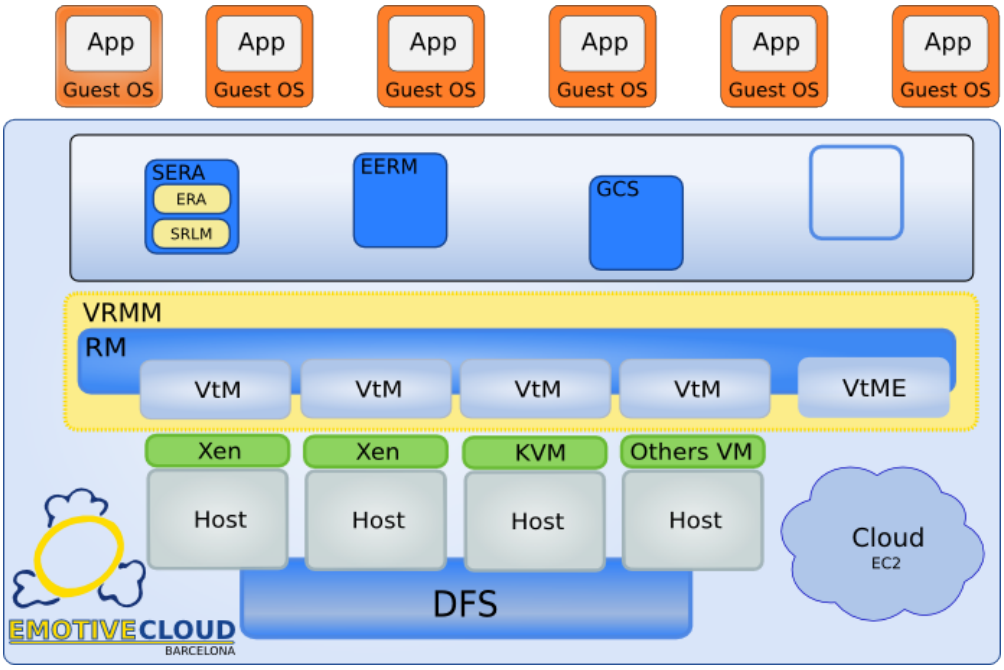


Figure 3.1: EMOTIVE structure

3.2.1 Data infrastructure

The data infrastructure makes EMOTIVE able to efficiently support different virtualization features such as migration and checkpointing. It distributes the data among the cluster nodes. It uses NFS in order to make the data of every node available from the other nodes. Thanks to this technique, Virtual Machines can be moved between nodes without losing connection. This capability allows new approaches such as consolidating the global system or giving more resources to a given application if the node is not able to do this locally. In addition, this data infrastructure allows each VM accessing data required by the user by using a shared repository also distributed among the nodes. It also allows storing data in the system in order to be reused later from other VMs.

3.2.2 Virtualized resource management

The first capability of the Virtualized Resource Management and Monitoring (VRMM) is the task of creating and maintaining virtual machine, which is done by the Virtualization Manager (VtM). This component is in charge of making this task as fast as possible in order to reduce the needed time to make the environment accessible to be used by the suer. In addition to the virtual machine life-cycle management, this layer takes profit of the resource management capabilities that are offered by the virtualization. This allows the system managing the resources of each task in a dynamic way. VMs are created on demand, according to the application requirements, both hardware (CPU type, amount of resources required) and software (required packages). These requirements are specified by means of the Open Virtualization Format (OVF). VRMM also provides an interface for monitoring the resource usage of each VM called Resource Monitor (RM). This components allow and efficient usage of the providers usages as well as supporting different quality of service levels according to the agreement reached with the user (SLA). With regard to the resource management infrastructure, this includes virtual machine migration in an efficient well and a checkpoint management system. On the one hand, migration allows moving a VM between different nodes in order to offer it more resources or to consolidate the overall system. On the other hand, checkpoint mechanisms makes the system able to be fail tolerant since it allows the system recovering tasks that where running in a broken machine. These can be recovered from other nodes that have enough resources.

Another feature of this layer is the task execution support. This mechanism makes the data required by a task to be executed available inside the VM in the moment the virtualized environment is up and running. This allows the execution of tasks at the very beginning of the VM life and it finally allows the recovering of the output data generated by the applications. This user data management also includes the capability of storing user data and the whole virtual machine configuration between different user sessions. For instance, the user can work with a virtual machine and when his work has finished this VM data can be stored in the system. The execution of a a VM with all the stored data and with the same state of the previous could be retaken in the future by a different node of the provider.

EMOTIVE also provide the capability to use external resource (public Cloud providers, such as Amazon EC2) by the VtME component which allows to create public, private and hybrid clouds.

3.3 Scheduler

A key issue of EMOTIVE is the global resource management, in other words, the global management of the different nodes of the system. For instance, choosing which node will execute each task or deciding the moment to create or migrate a VM in order to optimize the whole resources of the system. This is done by the scheduling layer which takes into account the overheads added in order to maintain and manage the virtualized environment. This layer is responsible for VMs placements and management during the execution with the possibility to use different policies and capabilities. In this sense, it can use a simplistic Round Robin algorithm or a consolidation-aware scheduling like Backfilling. The component responsible to manage the scheduling procedures is the DRP module which is a REST interface and implements methods (GET, POST, PUT and DELETE) used by HTTP protocol.

Nevertheless the scheduling policy to be used has to be decided before the whole system is compiled and running. It means that it is not possible to change scheduler or policy during the execution. The goal of this project is to change this behaviour in order to allow users to create, change and load different scheduling policies run-time, without the need to re-compile the whole source code. Main layers involved in this project are shown in figure 3.2.

In the next section there is a brief introduction about Web Services architecture and OCCI API used in EMOTIVE middleware and then new functionalities added to EMOTIVE with this project will be explained.

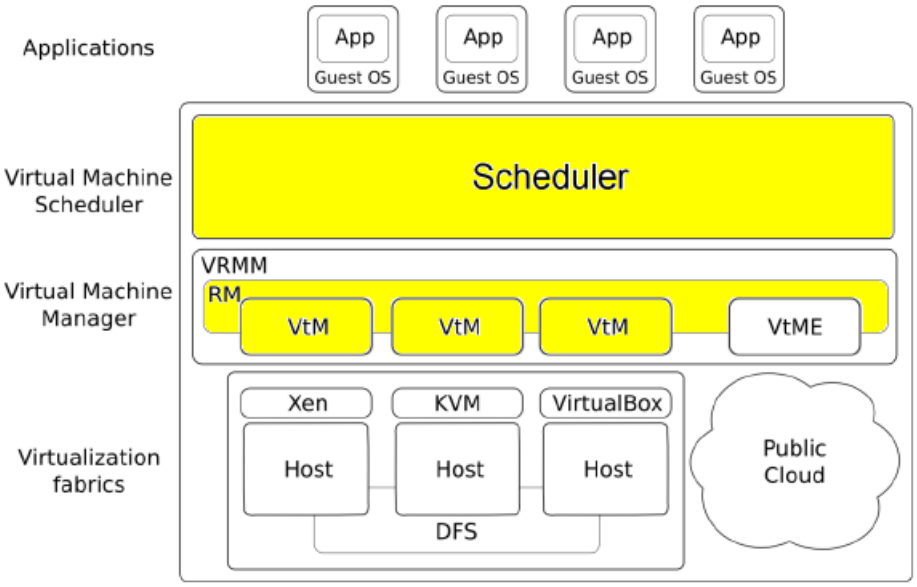


Figure 3.2: Relevant layers of this project

3.4 OCCI API and Web Services

EMOTIVE was originally designed using a distributed SOAP architecture but now it uses RESTful Web Services. This architecture allows the usage of only some parts of EMOTIVE and supports agile and dynamic construction of new Cloud environments. Its REST interface makes EMOTIVE highly interoperable with other Cloud solutions. Since different Cloud providers use their own and independent interface it is difficult to communicate and federate multiple providers. For this reason OCCI(Open Cloud Computing Interface) has been proposed as a common standard in order to overcome this problem.

OCCI is a RESTful Protocol and API for all kinds of management tasks and it acts as a service front-end to a providerâĂŹs internal management framework which uses HTTP methods, such as GET, POST, PUT and DELETE.

Methods used in EMOTIVE with a correspondence in the OCCI interface are shown boldfaced in table 3.1. Our interfaces basically allow:

- Compute: create, get, list and cancel Virtual Machines (supporting CIM and OVF).

- Network: similar to Compute methods but used to describe virtual networks.
- Jobs: used to submit jobs to Virtual Machines (we use JSDL format to describe them).
- Nodes: describes the system topology (used for EMOTIVE internals).

COMPUTE
- **String Env-ID = createEnvironment (Compute)**
- String Env-ID = createEnvironmentAndJob (Compute, JSDL)
- **terminateEnvironment (String Env-ID)**
- **List** $< Env - ID >$ **= getEnvironments ()**
- **Compute = getEnvironment (String Env-ID)**
- String state = getEnvironmentState (String Env-ID)
NODES
- String [Node-ID or Env-ID] = getLocation (String [Env-ID or Act-ID])
- List $< Node - ID >$ =getNodes ()
- nodeDown (String Node-ID)
- nodeUp (String Node-ID)
JOBS
- List $< Act - ID >$ = getActivities ()
- Act-ID = submitActivity (JSDL)
- cancelActivity (String Act-ID)
- String status = getActivityStatus (String Act-ID)
- List $< StringAct - ID >$ = getAllActivities ()
NETWORK
- **String Net-ID = createNetwork (Network)**
- **deleteNetwork (String Net-ID)**
- **Network = getNetwork (String Net-ID)**
- **List** $< Network >$ **= getListNetworks ()**
- String Net-ID = createVPN (Network)

Table 3.1: EMOTIVE REST API and OCCI methods

Snapshot 3.3 shows commands supported by EMOTIVE that have been implemented using previous methods and that can be used through console.

```
pctomeu:~$ help
HELP
    server:       change server to connect
                          + (hostname)
COMPUTE_____________________________________________________________________________
    policy:       change scheduling policy + file_name (file must be in the same folder of DRPClient)
    create:       create a new ovfDomEmo
    destroy:      destroy actual ovfDomEmo
                          + (name)  destroy a ovfDomEmo with the name
    new:          create a new ovfDomEmo
    show:         show ovfDomEmo
                          + all
    set:          set a property of the ovfDomEmo
                  General VM Attributes:
                          + (name/id/cpu/memory/taskid(unused)
                  Disk Section:
                         + (home/swap/extension)
                         + disk disk_name disk_href/"null" disk_capacity/"null"
                  Network Section:
                         + network net_name ip/"null" mac/"null" netmask/"null" gateway/"null"
    ls:           list all domains with their IPs.
    remove:        remove a disk/network from the ovfDomEmo
                          + disk/network name
NETWORK_____________________________________________________________________________
    net-show:     show nubaNet Object
                          + (all): show all networks in the DRP
                          + (id): show particular network in the DRP
    net-create:   create a new network
    net-destroy:  destroy network
    net-set:      set a property of the Network
                          + (name/address/netmask)
    exit:         quit this interactive terminal
```

Figure 3.3: EMOTIVE Console menu

3.5 EMOTIVE evolution

In order to develop an integrated policy framework which can manage and load different policies without stop services and recompile codes it has been necessary to modify the original EMOTIVE structure by adding a new layer, the Policy Manager, which is responsible to handle new scheduling operations that have to be used in the system.

New policies have to be defined following the rules of the new C-like language I developed, LEPIC (Language for Emotive Policies Integration and Creation) which will be loaded in the system thanks to a specific parser and compiler.

The new scheduling layers involved in policy management are organized as shown in figure 3.4.

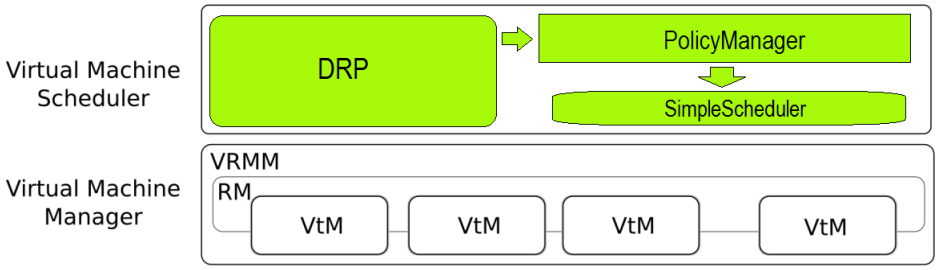


Figure 3.4: New scheduling layers

4

LEPIC - Language for Emotive Policies Integration and Creation

4.1 Policy concepts overview

EMOTIVE middleware provides users with virtualized environments where they can execute their tasks without any extra effort. These VMs, which aim to fulfill the user requirements in terms of software and system capabilities, are transparently managed by EMOTIVE in order to exploit the providerâĂ§s resources. EMOTIVE can easily be extended with multiple scheduling policies in order to manage the VMs using different criteria.

Therefore a policy is a rule that can be used to change the behaviour of a system. Separating policies from the managers that interpret them allows the behaviour and strategy of the management system to be changed without re-coding the managers. The management system can then adapt to changing requirements by disabling policies or replacing old policies with new ones without shutting down the system.

4.2 The compiler

In order to parse and to have the input represented in the form of an abstract syntax tree that defines the order of operations and allows us to traverse the different parts of the expression individually, SableCC, which is a Java-based tool that performs the same job as flex/bison, has been used in this project. It is a bottom up parser, which takes an unconventional and interesting approach of using object oriented methodology

for constructing parsers. This results in easy to maintain code for generated parser. In particular, generated frameworks include intuitive strictly-typed abstract syntax trees and tree walkers. SableCC also keeps a clean separation between machine-generated code and user-written code which leads to a shorter development cycle. The standard output from SableCC is Java code. Thus, the parser we're about to generate will be made into a number of Java files that we can incorporate into our own source code and extend them. Producing a compiler using SableCC requires the following steps:

1. Creating a SableCC specification file containing the lexical definitions and the grammar of the language to be compiled.
2. Launching SableCC on the specification file to generate a framework.
3. Creating one or more working classes (analyses, transformations on the AST (Abstract Syntax Tree) , or simply code generation classes) possibly inheriting from classes generated by SableCC.
4. Creating a main compiler class that activates lexer, parser and working classes.
5. Compiling the compiler with a Java compiler.

On output, SableCC generates files into four sub-packages of the specified root pack- age. The packages are named: *lexer*, *parser*, *node* and *analysis*. Each file contains either a class or an interface definition. The lexer package contains the Lexer and LexerException classes. These classes deal respectively with the generated lexer and the exceptions thrown in case of a lexing error. The parser package contains the Parser and ParserException classes. As expected, these classes are the parser and the exception thrown in case of a parsing errors. The node package contains all the classes defining the typed AST. The analysis package contains one interface and three classes. These classes are used mainly to define AST walkers.

The design of the SableCC compiler framework have a direct effect on the development cycle of a compiler because the debugging cycle is shorter than traditional compiler compilers. Since actions are written directly as Java classes, the source code being debugged was written by the programmer. This enables interactive debugging in an Integrated Development Environment.

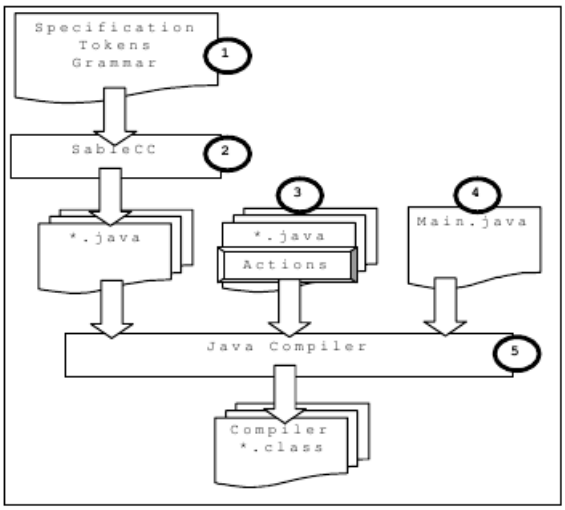


Figure 4.1: SableCC schema

4.3 Lexer

There are four sections in the specification file that influence the lexer generator of SableCC: the Package, Helpers, States and Tokens sections. On output, SableCC generates a lexer class whose behavior can be customized through inheritance, without modifying the generated code

4.3.1 Package

The Package declaration simply defines the name of the overall package. If this is excluded (which is valid according to SableCC) our namespace in the generated code will be blank and thus invalid.

Package net.emotivecloud.parser;

4.3.2 LEPIC Regular expressions, Helpers and Tokens

The syntax of regular expressions in SableCC is very similar to the formal definition of regular expressions. A regular expression can use character sets in addition to characters. Additionally, strings can also be used, and are specified with single quotes as

in: 'text'. SableCC regular expressions support union $regexp1|regexp2$, concatenation regexp1 regexp2 and Kleene closure regexp*, as well as parentheses (), and the ? $(a?b = ab|b = ab, b)$ and + $(a+ = aa* = a, aa, aaa, ...)$operators. There is no symbol for the empty string. The ? operator or an empty regular expression can be used to express this string $((a|)(a|) = aa, ab, ba, bb)$.

4.3.2.1 Helpers

Unlike other compiler compilers, SableCC has no support for macros. Instead, SableCC allows the use of helpers. A helper is a character set or a regular expression denoted by an identifier. Listing 4.1 shows Helpers defined in the Helpers section of the LEPIC specification.

Listing 4.1: LEPIC Helpers

```
Helpers

  all = [ 0 .. 127 ];
  digit = [ '0' .. '9' ];
  nondigit = [ '_' + [ [ 'a' .. 'z' ] + [ 'A' .. 'Z' ] ] ];
  digit_sequence = digit +;
  fractional_constant = digit_sequence ? '.' digit_sequence |
     digit_sequence '.';
  sign = '+' | '-';
  exponent_part = ( 'e' | 'E' ) sign ? digit_sequence;
  floating_suffix = 'f' | 'F' | 'l' | 'L';
  simple_escape_sequence = '\' ''' | '\"' | '\?' | '\\' | '\a' |
     '\b' | '\f' | '\n' | '\r' | '\t' | '\v';
  octal_digit = [ '0' .. '7' ];
  octal_escape_sequence = '\' octal_digit octal_digit ?
     octal_digit ?;
  hexadecimal_digit     = [ digit + [ [ 'a' .. 'f' ] + [ 'A' .. '
     F' ] ] ];
  hexadecimal_escape_sequence = '\x' hexadecimal_digit +;
  escape_sequence =     simple_escape_sequence |
     octal_escape_sequence | hexadecimal_escape_sequence;
  s_char = [ all - [ '"' + [ '\' + [ 10 + 13 ] ] ] ] |
     escape_sequence;
  s_char_sequence =     s_char +;
  nonzero_digit = [ '1' .. '9' ];
```

```
decimal_constant = nonzero_digit digit *;
octal_constant = '0' octal_digit *;
hexadecimal_constant = '0' ( 'x' | 'X' ) hexadecimal_digit +;
unsigned_suffix = 'u' | 'U';
long_suffix = 'l' | 'L';
integer_suffix = unsigned_suffix long_suffix ? | long_suffix
   unsigned_suffix ?;
c_char = [ all - [ ''' + [ '\' + [ 10 + 13 ] ] ] ] |
   escape_sequence;
c_char_sequence =     c_char +;
cr = 13;
lf = 10;
not_star = [ all - '*' ];
not_star_slash = [ not_star - '/' ];
apostrophe = 39;
tab = 9;
eol = cr | lf | cr lf;
```

When SableCC sees a helper identifier in a regular expression, it replaces it semantically (not textually) with its declared character set or regular expression.

4.3.2.2 Tokens

Tokens are defined in the Tokens section of the specification file with a syntax similar to helpers. For a given input, the longest matching token will be returned by the lexer. In the case of two matches of the same length, the token listed first in the specification file will be returned. It is also possible to define *Ignored tokens* that will be ignored when matched. For each token, SableCC generates a class in the package root.node with the name of the token as the class name. To be accurate, the exact name given to the class is computed from the name of the token by prefixing it with a capital 'T', replacing the first letter with its uppercase, replacing each letter prefixed by an underscore with its uppercase, and removing all underscores. For example, for a token named 'some token', SableCC would generate a class 'root.node.TSomeToken'. All token classes inherit from (or extend) the class 'root.node.Token'.

In listing 4.2 there are tokens and ignored tokens definitions for LEPIC language:

Listing 4.2: LEPIC Tokens

```
Tokens
```

```
dot   =       '.';
comma =       ',';
colon =       ':';
semicolon     =       ';';
l_par =       '(';
r_par =       ')';
l_bracket     =       '[';
r_bracket     =       ']';
l_brace       =       '{';
r_brace       =       '}';
star  =       '*';
div   =       '/';
mod   =       '%';
ampersand     =       '&';
plus  =       '+';
minus =       '-';
caret =       '^';
tilde =       '~';
excl_mark     =       '!';
quest_mark    =       '?';
bar   =       '|';
ellipsis      =       '...';
equal =       '=';
eq    =       '==';
neq   =       '!=';
lt    =       '<';
lteq  =       '<=';
gt    =       '>';
gteq  =       '>=';
arrow =       '->';
plus_plus     =       '++';
minus_minus   =       '--';
shl   =       '<<';
shr   =       '>>';
ampersand_ampersand   =       '&&';
bar_bar       =       '||';
star_equal    =       '*=';
div_equal     =       '/=';
mod_equal     =       '%=';
plus_equal    =       '+=';
```

```
minus_equal   =       '-=';
shl_equal     =       '<<=';
shr_equal     =       '>>=';
ampersand_equal       =       '&=';
caret_equal   =       '^=';
bar_equal     =       '|=';
if    =       'if';
else  =       'else';
while =       'while';
for   =       'for';
return        =       'return';

void  =       'void';
char  =       'char';
int   =       'int';
short =       'short';
long  =       'long';
float =       'float';
double        =       'double';
string =  'string';
const =       'const';
uri   =       'URI' | 'URL';
boolean = 'boolean';

floating_constant     =       fractional_constant exponent_part
   ? floating_suffix? | digit_sequence exponent_part
   floating_suffix?;
string_litteral       =       'L' ? '"' s_char_sequence? '"';
integer_constant      =       decimal_constant integer_suffix?
   | octal_constant integer_suffix? | hexadecimal_constant
   integer_suffix?;
character_constant    =       'L'? apostrophe c_char apostrophe
   ;
boolean_constant      =       'true' | 'false';
uri_constant = '"' ('http:\\' | 'vtm:\\') s_char_sequence'"';

comment       =       '/*' not_star* '*'+ ( not_star_slash
   not_star* '*'+ )* '/';
policy = 'policy';
policy_ext = 'policy_ext';
operation = 'operation';
```

```
  simple = 'simple';
  destroy = 'destroy';
  time_based = 'time_based';
  monitor = 'monitor';
  create = 'create';
  time = 'time';
  type = 'type';


  blank = (eol | tab | ' ');
  identifier    =       nondigit ( digit | nondigit )*;

Ignored Tokens
  blank,
  comment;
```

4.3.3 Productions

SableCC supports an EBNF-like syntax for productions but in the specification there is no place for action code, which means code to be executed every time a production is recognized.

Every production class is automatically named by using the production name, surrounded by braces, prefixing it with an uppercase 'P', replacing the first letter with an uppercase, replacing each letter prefixed by an underscore with an uppercase, and removing all underscores. For instance, production *compound_ statement* will result in class *PCompoundStatement*. If a production has different alternatives, the alternative class is named like its production class, but the uppercase 'P' prefix is replaced by an uppercase 'A' and followed by the production name which generates it. For instance *call_ expression* production, generated by *basic_ statement* will result in class *ACallExpressionBasicStatement*.

SableCC does not give direct access to element variables. Instead, it provides accessor methods. Accessors are *getxxx* and *setxxx* methods, where the *xxx* is the name of the element (with the usual uppercase transformation). SableCC uses these accessors to further prevent the construction of an invalid AST.

Productions are listed in listing 4.3:

Listing 4.3: LEPIC Productions

```
Productions

  program = policy_def functions*;
  policy_def = {non_empty_name} policy identifier semicolon | {
     empty};
  type_specifier =      {void}  void |
                        {char}  char |
                        {int}   int |
                        {float} float |
                        {uri}    uri |
                        {string} string |
                        {boolean} boolean;


  functions = type fn_decl;
  fn_decl = {simple_create} simple identifier? l_par operation
     create r_par function_body_simple|
            {simple_destroy} simple identifier? l_par operation
               destroy r_par function_body_simple|
            {monitor_time} monitor identifier? l_par time
               integer_constant r_par function_body_monitor|
            {monitor_notime} monitor identifier? l_par r_par
               function_body_monitor;


  function_body_simple  = l_brace variable_declaration* statement
     * stop_statement r_brace;
  function_body_monitor = l_brace variable_declaration* statement
     * r_brace;

  variable_declaration  =  {declaration} type_specifier
     identifier additional_declarator* semicolon |
                           {assign_decl} type_specifier
                              identifier equal constant
                              additional_declarator* semicolon;

  additional_declarator = {additional} comma identifier |
                          {additional_assign} comma identifier
                             equal constant;
```

```
statement =   {comp_stmt}     compound_statement |
              {basic_stmt}    basic_statement semicolon |
              {if}    if l_par conditional_expression r_par
                 compound_statement |
              {if_else}       if l_par conditional_expression
                 r_par [then_comp_stmt]:compound_statement else
                  [else_comp_stmt]:compound_statement |
              {while} while l_par conditional_expression r_par
                 compound_statement |
              {for}   for l_par [start]:basic_statement ? [
                 sc_one]:semicolon conditional_expression ? [
                 sc_two]:semicolon [iter]:basic_statement ?
                 r_par compound_statement;

compound_statement    = l_brace statement* r_brace;
basic_statement       =       {call_expression}
   call_expression |
                               {modify_expression}
                                  modify_expression |
                               {short_expr}       short_expr;




call_expression       =       {id_expr} identifier l_par
   arglist? r_par|
                               {bin_expr} identifier l_par
                                  binary_expression r_par ;

arglist  =    {value} value arglist_tail*| {expr}
   binary_expression;
arglist_tail  = comma value;

modify_expression     =       identifier equal rhs;


short_expr            =       {pp} identifier plus_plus |
                               {mm} identifier minus_minus;

simple_expression     =  {id} identifier |
                         {constant}   constant;
```

```
conditional_expression = {rel}        [left]:value relop [right
   ]:value |
                         {value}      value;


relop =        {eq}    eq |
               {neq}   neq |
               {lt}    lt |
               {lteq}  lteq |
               {gt}    gt |
               {gteq}  gteq;

value =        {identifier}    identifier |
               {constant}      constant;


stop_statement  =     {return_value}  return value semicolon |
                      {return_function_value} return
                         call_expression semicolon;

arrayref       =      identifier reflist +;
reflist        =      l_bracket value r_bracket;



constant = {floating} unop? floating_constant |
           {string}   unop? string_litteral |
           {integer}  unop? integer_constant |
           {character}unop? character_constant |
           {uri}   uri_constant |
           {boolean} boolean_constant;

unop  =        {plus}  plus |
               {minus} minus |
               {tilde} tilde |
               {excl_mark}     excl_mark;

rhs   =        {binary} binary_expression |
               {short}  short_expr |
               {unary}  unary_expression;
```

```
  binary_expression  = [l_value]:value binop [r_value]:value ;
  binop =       {div}   div |
                {mod}   mod |
                {plus}  plus |
                {minus} minus |
                {star}  star;

  unary_expression =    {simple}        simple_expression |
                        {call}  call_expression ;
```

4.3.4 Generating the parser

Once the grammar is defined, it is necessary to compile and to run the SableCC file containing the previous sections. It will result in the following output:

```
 -- Generating parser for lepic_grammar.scc in /home/mauro/
    Emotive_code/branches/mauro/Scheduler/LepicParser/src/main/
    java
Adding productions and alternative of section AST.
Verifying identifiers.
Verifying ast identifiers.
Adding empty productions and empty alternative transformation if
   necessary.
Adding productions and alternative transformation if necessary.
computing alternative symbol table identifiers.
Verifying production transform identifiers.
Verifying ast alternatives transform identifiers.
Generating token classes.
Generating production classes.
Generating alternative classes.
Generating analysis classes.
Generating utility classes.
Generating the lexer.
 State: INITIAL
 - Constructing NFA.
..........................................................
 - Constructing DFA.
..........................................................
..........................................................
 - resolving ACCEPT states.
```

```
Generating the parser.
..........................................................
..........................................................

Inlining
..........................................................
..........................................................
```

In the generated directory, there will be five packages:

- net.emotivecloud.parser
- net.emotivecloud.analysis
- net.emotivecloud.lexer
- net.emotivecloud.nodes
- net.emotivecloud.parser

These packages contain classes in the net.emotivecloud.parser namespace that will be modified in order to generated the code we need.

4.4 Policy definition

Observing the rules of the grammar defined in the previous sections is possible to write policies with the schema shown below (editable parts are those in upper case)[1] :

```
policy POLICY_NAME;
type TYPE(operation OPERATION){

        VARIABLES_DECLARATION;
        CODE;

return RETURN_STATEMENT;
}
```

4.4.1 Policy name

A name can be assigned to the policy defined (*policy POLICY_NAME*). If this line is not present the policy will not have any name.

[1] *operation* statement is present only in *simple* policies

4.4.2 Type

With LEPIC language three different types of policy can be defined (details for each type will be describe in the following sections):

- **Simple**: allow user to define new operations for creating and destroying virtual machines.

  type simple

- **Monitor**: user can set actions that have to be performed when some conditions are satisfied (e.g.: when the load of a node exceed a threshold perform an action).

  type monitor

  It is also possible to set the time interval for checking conditions (default 500 ms) and moreover we can define multiple monitoring parts in the same policy. In this way we can perform actions with different time intervals: E.g :

  ```
  policy monitor_p;
  type monitor Part1 (time 500){...}
  type monitor Part2 (time 1000){...}
  ```

  An example is shown in section 4.4.7.2: Monitoring Policy.

- **Extended**: it is possible to define a policy that contains both simple and monitoring parts.

4.4.3 Operation

The definition of an operation is present only in simple policies. User can choose for two different operations:

- **create**: when a creation of a VM is requested it will be create in the node returned in the RETURN_STATEMENT;

- **destroy**: using this option, when a destroy operation is requested it will turn off the VM whose Id is returned in the RETURN_STATEMENT

4.4.4 Variables declaration

Variable declaration follows a C-like syntax and the initialization can also be done in this section. Types allow in LEPIC are the following:

- void
- char: initialization surrounded by single quotes. E.g.: *char var = 'a';*
- int: E.g.:*int num = 0;*
- boolean: it can have *true* or *false* value. E.g.: *boolean b = false;*
- float E.g.:*float fnum = 1.0;*
- string: initialization surrounded by double quotes. E.g *string vm = "id";*
- uri: initialization surrounded by double quotes and it has to be an http URI. E.g.: *URI node = "http://localhost"*

4.4.5 Code

In this section is possible to use some C-like statements. They are listed in the following lines with some examples.

- **Variable assignment**:

```
var = value;
```

- **Expressions**:

```
var = 5 + 10;
```

- **If**:[1]

```
if (condition) {
        statement;
        }
```

- **If-else**:

[1] unlike C language, braces must be always present even if there is only one statement

```
if (condition) {
        statement;
        }
else {
        statement;
        }
```

- **While**:

```
while (condition) {
        statement;
        }
```

- **For**:

```
for (initialization; condition; increment) {
        statement;
        }
```

Note that if a variable is initialized in the *variables declaration* section that value will be assigned to the variable only the first time the policy is loaded. Contrariwise if we need that a variable must have an initial value every time the policy runs we have to initialize it in the code section using a *variable assignment*.

LEPIC language provides also some functions (table 4.1) that allow to perform tasks or to retrieve information from the system whose implementations is reported in section 5.3.3: EMOTIVE API in LEPIC.

4.4.6 Return statement

Return statement is used only in simple policy definition and it returns the value to the Policy Manager which will perform the defined operations.

FUNCTION	RETURNED VALUE
int size();	number of active nodes in the cloud
URI getURI(int i);	URI of the node i
int numVMnode(int/URI i);	number of VMs running in the node i
float freeCPU(int/URI i);	percentage of free CPU in the node i
void destroyVM(String s);	it does not return any value. It destroys the VM whose id is s
boolean exists(string id);	true if the node or the VM passed as parameter is present in the cloud
int numVMs(int i);	the number of VMs stored in the node i
string getVMId(int i);	VM id whose index in the domains list is i
float getVMCPU(int/string i);	amount of used CPU by the VM whose id or index is i
int getVMMem(int/string id);	amount of memory needed by the VM whose id or index is i
int freeMem(int/URI i);	free memory in the node whose URI or index in the list of nodes is i

Table 4.1: EMOTIVE API in LEPIC

4.4.7 Examples

4.4.7.1 Simple Policy

In order to demonstrate how a simple policy can be defined, a Round Robin algorithm implemented with LEPIC is shown in listing 4.4:

Listing 4.4: Simple policy: Round Robin

```
policy policy_RR;
type simple(operation create){

URI nodeToSelect;
int lastSelected=0;
int size;

    size = size();
    nodeToSelect = getURI(lastSelected % size);
    lastSelected = lastSelected +1;
```

```
    return nodeToSelect;

}
```

An example of a policy dealing with *destroy* operation is reported in listing 4.5; When "destroy" is typed in the console this policy compute the CPU used by each VMs in the system and destroy the one which has higer CPU load;

Listing 4.5: Simple policy: operation destroy

```
policy destroy_example;
type simple(operation destroy){

int i, n, index;
float cpu, min;
string d;

  index = 0;
  min = 0.0;
  n = numVMs();

  for (i=0; i < n; i++){
    cpu = getVMCPU(i);
    if (cpu > min){

      index = i;
      min = cpu;
    }
  }

  d = getVMId(index);

return d;
}
```

4.4.7.2 Monitoring Policy

When a monitoring policy is loaded it start to run. When the conditions written in the code are satisfied, defined actions are performed. It is also possible to give a name to the monitor function as shown below (*CheckMem()*). Sample policy in listing 4.6 causes

the destruction of the VM with id *vm4* when the CPU load of the node *0* is lower than *380.00*. This condition is checked every 1000 ms.

Listing 4.6: Moniotring policy

```
policy policyMonitorCPU;
type monitor CheckMem(time 1000){

string vm = "vm4";
float num;
float c = 380.00;

        num = freeCPU(0);
        if (num < c)
            {
              destroyVM(vm);
            }

}
```

4.4.7.3 Extended Policy

An extended policy is made up of both simple and monitoring policies. Listing 4.7 shows as it can be defined. That is simply a union between the previous policies. When this is loaded the monitoring part will keep on running, performing actions when conditions are satisfied, while the simple part is only executed when the operation is called, in this case *create*.

Listing 4.7: Extended policy

```
policy policy_Ext;
type simple(operation create){

URI nodeToSelect;
int lastSelected=0;
int size;

    size = size();
    nodeToSelect = get(lastSelected % size);
    lastSelected = lastSelected +1;
```

```
    return nodeToSelect;

}
type monitor CheckMem(){

string vm = "vm4";
float num;
float c = 380.00;

        num = freeCPU(0);
        if (num < c)
            {
              destroyVM(vm);
            }

}
```

4.5 Parsing and code generation with LEPIC

Once a policy is defined in a text file (*FileName*), it can be loaded in the EMOTIVE middleware by typing in the client *policy FileName*. In order to parse it, it has been necessary to develop some JAVA modules:

- LepicCompiler.java
- SemanticAnalyzer.java
- EmotiveAPI.java
- NumberOperation.java

4.5.1 LepicCompiler

Lepic Compiler is stack based so to translate an assignment or an operation statement all the factors have to be pushed on the stack and then we pop the result into the variable.

All the variables are stored in an data structure (HashMap) that will be used to save or retrieve variables values. In order to implement the code generator we need to

do a depth first traversal of the tree. This is made by overwriting methods generated by SableCC. The DepthFirstAdapter is a class auto generated by SableCC. It allows to easily traverse the generated AST depth first, while giving various hook points along the way. If we want to add some action code when visiting a node of type Xxx in a derived class, we must override the caseXxx method and add into it the tree walking code. Moreover each node in the tree has an *In* and *Out* method that can be override. *In* is called before the children are traversed while *Out* is called after the children have been traversed.

For instance we encounter an identifier it is pushed in the stack

Listing 4.8: LepicCompiler - method inAIdentifierValue

```
public void inAIdentifierValue(AIdentifierValue node) {
    Object v = hm.get(node.getIdentifier().getText());
    stack.push(v);
}
```

If we are parsing the variable declaration section, values have to be stored in the HashMap in order to be used later. For instance, the *outAAssignDeclVariableDeclaration(AAssignDeclVariableDeclaration node)* method, after checking the declared type of the identifier (*PTypeSpecifier t*) stores its value in the HashMap. A LinkedList is also used in order to store additional declarators or function arguments.

Listing 4.9: LepicCompiler - method outAAssignDeclVariableDeclaration

```
public void outAAssignDeclVariableDeclaration(
   AAssignDeclVariableDeclaration node) throws
   NumberFormatException{

        PTypeSpecifier t = node.getTypeSpecifier();
        this.arguments.addLast(node.getIdentifier().getText());

        Iterator iterator = arguments.iterator();
        Object b = stack.pop();

            if (t instanceof AIntTypeSpecifier) {

                if (b instanceof Integer) {
                    hm.put(iterator.next().toString(), b);
```

```
            } else {
                throw new NumberFormatException("Wrong types
                    in variable declarations");
            }
        }
        if (t instanceof ACharTypeSpecifier) {
             if (b instanceof Character) {
            hm.put(iterator.next().toString(), b.toString().
                charAt(0));}
            else {
                throw new NumberFormatException("Wrong types
                    in variable declarations");
            }

        }

        if (t instanceof AFloatTypeSpecifier) {
            if (b instanceof Float) {

            hm.put(iterator.next().toString(), Float.
                parseFloat(b.toString()));}
            else {
                throw new NumberFormatException("Wrong types
                    in variable declarations");
            }

        }

        if (t instanceof AUriTypeSpecifier) {
            try {
                hm.put(iterator.next().toString(), new URL(b.
                    toString().replace("\"", "")));

            } catch (MalformedURLException ex) {
                Logger.getLogger(LepicCompiler.class.
                    getSimpleName()).log(Level.SEVERE, null,
                    ex);
            }

       }
        if (t instanceof AStringTypeSpecifier) {
```

```
                if (b instanceof String) {
                hm.put(iterator.next().toString(), b.toString().
                    replace("\"", ""));}
                else {
                    throw new NumberFormatException("Wrong types
                        in variable declarations");
                }

            }

        }
        arguments.clear();
    }
```

The implementation of LepicCompiler follows this approach:

- it performs operations and modifies variables values
  - *public void outARelConditionalExpression(ARelConditionalExpression node)*
  - *public void outAModifyExpression(AModifyExpression node)*
- it handles *if*, *if-else*, *for* and *while* statements
  - *public void caseAIfStatement(AIfStatement node)*
  - *public void caseAIfElseStatement(AIfElseStatement node)*
  - *public void caseAForStatement(AForStatement node)*
  - *public void caseAWhileStatement(AWhileStatement node)*
- it allows to call functions
  - *public void outACallExpressionBasicStatement*

If we are parsing a simple policy a value is returned at the end of the compilation and it can be read by the *public Object getRetVal()* method.

4.5.2 SemanticAnalyzer

As we wote before, once policy is created observing LEPIC rules, it is necessary to ensure that the statements defined are grammatically correct. This task is performed by the SemanticAnalyzer module where the semantic analysis is processed by a depth first traversal on the abstract syntax tree, storing information about identiiňĄers in a symbol table and checking for errors.

Listing 4.10: SemanticAnalyzer - check for errors

```
public void inADeclarationVariableDeclaration(
    ADeclarationVariableDeclaration node) {
           if (!checkIdentifier(node.getIdentifier().getText())) {
             Logger.getLogger(SemanticAnalyzer.class.
                getCanonicalName()).log(Level.SEVERE, "Identifier
                '" + node.getIdentifier().getText() + "' already
                defined");
             state = false;
         } else {
             this.hm.put(node.getIdentifier().getText(), 0);
         }
      }

    public boolean checkIdentifier(String key) {
         if (this.hm.containsKey(key)) {
             return false;
         } else {
             return true;
         }
     }
```

A boolean variable (*private boolean state*) is used in order to inform the Policy Manager if a syntax error has occurred.

4.5.3 EmotiveAPI

This class contains functions that can be used in the policy definition.

Listing 4.11: Emotive API in LEPIC

```
public Object FunctionHandler(String name, LinkedList args)
   throws UnsupportedOperationException{

    if (name.equalsIgnoreCase("size")) {
        if (args.isEmpty()) {
            return getSize();
        } else {
            throw new UnsupportedOperationException("Wrong
               number of parameters in function 'size()'");
        }

    } else if (name.equalsIgnoreCase("getURI")) {
        if (args.size() == 1) {

            try {
                Integer i = Integer.parseInt(args.get(0).
                   toString());
                return getURINode(i);
            } catch (NumberFormatException e) {
                Logger.getLogger(EmotiveAPI.class.
                   getSimpleName()).log(Level.SEVERE, "Wrong
                   parameter in function 'get(int i)'");
            }
            return null;
        } else {
            throw new UnsupportedOperationException("Wrong
               number of parameters in function 'get(int i)
               '");
        }
    } else if (name.equalsIgnoreCase("numVMnode")) {
        if (args.size() == 1) {
            try {
                return this.getNumVMnode(args.get(0).toString
                   ());
            } catch (NumberFormatException e) {
                Logger.getLogger(EmotiveAPI.class.
                   getSimpleName()).log(Level.SEVERE, "Wrong
                   parameter in function 'get(int i)'");
            }
            return null;
        } else {
```

```
                throw new UnsupportedOperationException("Wrong
                    number of parameters in function 'get(int i)
                    '");
            }


        } else if (name.equalsIgnoreCase("freeCPU")) {
            if (args.size() == 1) {
                try {
                    return freeCPU(args.get(0).toString());
                } catch (NumberFormatException e) {
                    Logger.getLogger(EmotiveAPI.class.
                        getSimpleName()).log(Level.SEVERE, "Wrong
                        parameter in function 'get(int i)'");
                }
                return null;
            } else {
                throw new UnsupportedOperationException("Wrong
                    number of parameters in function 'get(int i)
                    '");
            }

        }
                .
                .
                .
```

In listing 4.11 we can see a part of method *FunctionHandler(String name, LinkedList args)*, which checks if the called function is present in LEPIC and if number and type of possible parameters are correct it performs the mapped operations, implemented in the Policy Manager.

4.5.4 NumberOperation

When a operation is required it is necessary to check if two operands have the same type in order to perform it. For instance if we want to perform a multiplication method in listing 4.12 will be called:

Listing 4.12: NumberOperation - multiplication

```
private Number mul(Number a, Number b) throws ArithmeticException
    {
        if (a instanceof Integer && b instanceof Integer) {
            return (Integer) a * (Integer) b;
        } else if (a instanceof Float && b instanceof Float) {
            return (Float) a * (Float) b;
        } else {
            throw new UnsupportedOperationException("Wrong
               operands type");
        }

    }
```

NumberOperation class also handles relational operators.

Listing 4.13: NumberOperation - relational operators

```
public boolean relop(Object a, String op, Object b) {


        if (op.equalsIgnoreCase("==")) {
            if (a.equals(b)) {
                return true;
            } else {
                return false;
            }
        } else if (op.equalsIgnoreCase("!=")) {
            if (a.equals(b)) {
                return false;
            } else {
                return true;
            }
        } else if (op.equalsIgnoreCase("<")) {
            return this.lowerThan(a,b);
        }
        else if (op.equalsIgnoreCase(">")) {
            return this.greaterThan(a,b);
        }
        else if (op.equalsIgnoreCase("<=")) {
            return this.lowerEqualsThan(a,b);
        }
        else if (op.equalsIgnoreCase(">=")) {
```

```
            return this.greaterEqualsThan(a,b);
        }
        else {
            throw new UnsupportedOperationException("Wrong
               operands type");
        }
    }
```

5

Implementation of policy management framework in EMOTIVE supporting LEPIC

In this section we will describe changes and developments that have been made to the EMOTIVE middleware in order to implement the policy management framework supporting LEPIC.

5.1 DRP Client

In order to interact with EMOTIVE when we want to change scheduling policy it has been necessary to modifiy the DRPClient module which interface to the middleware. EMOTIVE runs by default with the SimpleScheduler policy but when a new policy contained in a file has to be uploaded, we simply need to type in the client the following command:

policy File_Name

In order to handle this command *Console* class has been modified by adding the code in listing 5.1.

Listing 5.1: New command in Console

```
else if (parsedCMD.startsWith("policy")) {
        try {
```

```
                String[] params = command.split(" ");
                if (drp == null)
                        drp = new DRPClient(server);
                if (params.length == 2)
                        {
                        String file = params[1];
                        if (file.equalsIgnoreCase("
                           simpleScheduler"))
                                user_policy = false;
                        else
                                user_policy = true;
                        drp.upload(file);
                        }
                else
                        {
                        System.out.println("ERROR: Unknown
                           command.");
                        }
                } catch (Exception e) { //DRPException
                    System.err.println("   DRP Error: " + e.
                       getMessage());
                }
```

File with the new policy will be uploaded thanks to the *public void upload(String fileName)* method in the *DRPClient* class, properly modified (listing 5.2).

Listing 5.2: DRPClient: Upload method

```
public void upload(String fileName) {

    if (fileName.equalsIgnoreCase("simpleScheduler")){
        this.changePolicyToScheduler();
    }
    else{

     try {
         String p = System.getProperty("user.dir")+"/"+
            fileName;

         FileInputStream stream = new FileInputStream(new File
            (p));
```

```
            FormDataMultiPart part = new FormDataMultiPart().
               field("file", stream, MediaType.
               MULTIPART_FORM_DATA_TYPE);

            WebResource resource = client.resource(this.
               getAddress()).path("policy").path(fileName);
            resource.type(MediaType.MULTIPART_FORM_DATA_TYPE).
               post(String.class, part);

                stream.close();
                System.out.println("File '"+fileName+"' uploaded
                   and the new policy is running...");
            } catch (IOException ex) {
                Logger.getLogger(DRPClient.class.getName()).log(
                   Level.SEVERE, null, ex);
                System.out.println("");
                System.out.println("Place your policy file in "+
                   System.getProperty("user.dir")+"/");
                System.out.println("");
            }
        }
    }
```

The client side application constructs an outgoing multipart/form-data message containing the policy and it will be sent to the DRP server side (*PATH: /policy/fileName*). To issue requests, it is necessary to create a WebResource object, which encapsulates a web resource for the client. Using it we can build build requests to send to the web resource and to process responses returned from it. If we want the system back to the default scheduler, we can just type in the console the following command:

policy SimpleScheduler

The following method will communicate to the server side that the simple scheduler has to be loaded (PATH: */policy/simpleScheduler*):

Listing 5.3: Load Simple Scheduler

```
    public void changePolicyToScheduler(){
        String ret;
        try {
```

```
            WebResource resource = client.resource(this.
                getAddress()).path("policy").path("
                simpleScheduler");
            ret = resource.type(MediaType.APPLICATION_XML).post(
                String.class, "simpleScheduler");
            System.out.println(ret);
        } catch (Exception ex) {
            System.err.println("[DRPClient]: Error loading
                SimpleScheduler");
            ex.printStackTrace();
        }
    }
```

5.2 DRP

Unlike original EMOTIVE architecture, *DRP* class extends *Policy Manager*, which is the layer whose main task is to manage new scheduling operations. When DRP receives a request in order to change policy it creates and writes a new file containing its definition calling the following method:

> private void writeToFile(InputStream uploadedInputStream, String uploadedFileLocation);

Then through *public Response handleUpload(@FormDataParam("file") InputStream stream, @PathParam("fileName") String fileName, @Context HttpServletRequest req)* it sets the new policy and parses it for the first time, storing variables in a new HashMap.

Listing 5.4: Part of DRP handleUpload() method

```
        .
        .
        .
super.setCurrent_policy(uploadedFileLocation);
HashMap<String, Object> hmap = new HashMap<String, Object>();
        .
        .
        .
super.setFirstTime();
super.runParser(hmap);
```

5.3 Policy Manager

Policy Manager is the most important layer of the scheduling process. It is in charge of implementing the LEPIC language, loading new policies and providing the user with new functionalities.

PolicyManager class extends the *Scheduler* class. In this way if an operation is not defined in the new loaded policy the default one in the *SimpleScheduler* will be used, avoiding errors or system crashes.

In order to parse a new policy uploaded to the server side, the $runParser(HashMap < String, Object > hm)$ method is called and it works as follows:

The first time that the file has to be parsed, its content is stored in a String and splitted depending on the policy type.

```
if (this.isFirstTime == true) {
        .
        .
        .
    FileInputStream stream = new FileInputStream(new File(
        this.current_policy));
    FileChannel fc = stream.getChannel();
    MappedByteBuffer bb = fc.map(FileChannel.MapMode.
        READ_ONLY, 0, fc.size());

    // Use a decoder and store policy
    String s = Charset.defaultCharset().decode(bb).toString()
        ;
    stream.close();
```

That is because if in case of a monitoring policy we can handle simple and monitor parts using two different methods:

- *public void simplePolicyManager(Parser parser, LepicCompiler c)*: responsible for *simple* part
- *public void monitorPolicyManager(String p)*: responsible for *monitoring* part

Once the new policy is stored in a string, we check for semantic errors (using the SemanticAnalyzer class of LEPIC package): if no errors are detected we divide simple

and monitor parts of the policy (if present) in a vector(*vett_ policy[i]*). This will allow us to handle the two types independently with the previous methods.

```
if (checkVariables(s)) {
        this.vett_policy = s.split("type");

        for (i = 0; i < this.vett_policy.length; i++) {
            if (this.vett_policy[i].startsWith(" simple")) {
                this.index_simple = i;
            } else if (this.vett_policy[i].startsWith(" monitor")
               ) {
                this.index_monitor[num_monitor] = i;
                    num_monitor++;
            }
        }
}
```

Making this division it is easier to parse the policy focusing on its different sections. As it can be noticed *num_ monitor* index is incremented every time we have a monitoring part. This is because by using this framework it is possible to have multiple monitoring operations (max 10) using different time intervals for each of them.

5.3.1 Simple policy management

Like most compiler-compilers, SableCC splits the work into a lexer and a parser. The lexer reads in characters and chunks them into tokens as defined by the Tokens section of the grammar file. In case of a simple policy, the following code shows how it is handled:

- *lexer* object is created in order to scan the original text file.
- the *Parser* class is used to build the AST.
- A new *LepicCompiler* object, which provides a depth-first traversal of the AST, is created and given as a parameter to the *simplePolicyManager()* method:

```
    StringReader stringReader = new StringReader("type " +
       this.vett_policy[this.index_simple]);
    PushbackReader reader = new PushbackReader(stringReader);
Lexer lexer = new Lexer(reader);
Parser parser = new Parser(lexer);
```

```
this.hmap = hm;

LepicCompiler c = new LepicCompiler(this.hmap, this.
    isFirstTime, this);
     simplePolicyManager(parser, c);
```

simplePolicyManager(parser, c) performs the parsing operation applying the *LepicCompiler*, updates the HashMap with the new values, after having setted the operation type of the policy, and it will get the defined return value.

```
Start ast;
ast = parser.parse();
ast.apply(c);

if (op.equalsIgnoreCase("create"))
        this.isCreate = true;
else if (op.equalsIgnoreCase("destroy"))
        this.isDestroy = true;
this.hmap = c.getHM();
this.retval = c.getRetVal();
```

Once the new policy has been uploaded, parsed and compiled it is ready to perform the scheduling operation following its approach. The following subsections show how create and destroy operations are managed.

5.3.1.1 Operation create

In a simple policy, performing an operation *create*, we define the node in which a VM has to be created. The URI of the node is returned by the LEPIC *return* statement and it is stored in the variable *retval*.

When the command "*create*" is typed in the console, the Policy Manager works as follows:

1. it checks if the default scheduler is setted; in this case the default operation is performed (defined in the *Scheduler* class)

```
if (this.simplescheduler)
     return super.create(ovf, jsdl);
```

2. it checks if this policy manages the operation *create*; if true, the policy is parsed: unlike the loading of the policy performed the first time, during this process

variables values are computed and used in order to determine the node for VM placement. If the return value is null, the node is chosen by using the default scheduler as above.

```
if (this.isCreate) {
        this.runParser(this.hmap);
        if (this.retval == null) {
                Logger.getLogger(PolicyManager.class.
                   getSimpleName()).log(Level.INFO,"Node to
                   Select is NULL: using Simpe Scheduler");
                return super.create(ovf, jsdl);
                }
```

3. if the return value has not a *null* value, the URI is created by parsing the variable *retval* and the VM is created in the desired node. If a URISyntaxException occurs, the default scheduler is used.

```
try {
        URI nodeToSel = new URI(this.retval.toString());
        Logger.getLogger(PolicyManager.class.getSimpleName())
           .log(Level.INFO,"Policy '"+this.policy_name+"'.
           Creating VM in node: " + nodeToSel);
        return super.create(ovf, jsdl, nodeToSel);

         } catch (URISyntaxException ex) {
                Logger.getLogger(PolicyManager.class.
                   getSimpleName()).log(Level.SEVERE,"Wrong
                   URI for destination node: using Simpe
                   Scheduler");
                return super.create(ovf, jsdl);
        }
```

5.3.1.2 Operation destroy

Original EMOTIVE allows to destroy a VM passing its id as a parameter.

A LEPIC policy which has an operation *destroy* definition gives us the opportunity to compute the VM that must be destroyed when in the console is "typed *destroy*" (without any parameter).

As for the *create* operation the id of the VM to be destroyed is returned by the compiler parsing and executing the policy (*retval*).

When "*destroy*" is typed in the console without any parameter DRPClient module send to the server side DRP a string (*"user_defined_destroy_policy"* which denotes that the VM to destroy has to be computed running the loaded policy and passed to the destroy method implemented in the Scheduler class (*super.destroy(retval.toString(), userDN);*). In case the command is followed by a string, which represent the VM id, EMOTIVE will give this parameter to the method as the original EMOTIVE destroy operation.

```
if (vmId.equals("user_defined_destroy_policy") && !this.
    simplescheduler && this.isDestroy) {

        runParser(this.hmap);
        if (retval == null) {
            Logger.getLogger(PolicyManager.class.
                getSimpleName()).log(Level.INFO, "Impossible
                to destroy VM: Null value returned");
        } else{
            super.destroy(retval.toString(), userDN);
        }
    } else {
        super.destroy(vmId, userDN);
    }
```

5.3.2 Monitoring policy management

Thanks to its structure, the *PolicyManager* layer has the ability to handle more than one event with different interval times. Since we can have different monitoring actions and they are stored in an array, we manage each of these parts by the method *public void monitorPolicyManager(String p)*.

```
if (index_monitor[0] != -1) {
                for (int j=0; j < num_monitor; j++){
                    monitorPolicyManager(this.vett_policy[
                        index_monitor[j]]);
                }
```

In order to handle different monitoring parts, a Java *ExecutorService* has been used. It provides a powerful framework for asynchronous task execution which abstracts away many of the complexities associated with the lower-level abstractions like raw Thread.

We use a thread pool that reuses a fixed number of threads equals to the number of monitoring parts we have to handle.

```
executor = Executors.newFixedThreadPool(num_monitor);
```

So when the policy is loaded, Runnable task is submitted for execution and it runs until a new policy is loaded.

```
public void monitorPolicyManager(String p) {
    if (this.isFirstTime) {
        this.monitor_running = true;
        executor.execute(new Monitor(this, p));
    }
}
```

The *Monitor* class is responsible for managing each operation setted in the policy. It parses the section containing the code that has to be performed and gets the value representing the time interval for the sleeping time.

```
while (!Thread.currentThread().isInterrupted()) {
            try {
                parseMonitor(this.hmap);
                System.out.println("Function: "+ fn_name);
                System.out.println("Sleeping time " + time + " ms
                    ");
                Thread.sleep(time);
            } catch (InterruptedException ex) {
                Thread.currentThread().interrupt();
            }
        }
```

When a new policy is loaded it necessary to stop all the running threads which were performing monitoring operations.

```
        executor.shutdownNow();
        executor.awaitTermination(50, TimeUnit.MILLISECONDS);
```

5.3.3 EMOTIVE API in LEPIC

In this section we will describe the implementation of LEPIC functions that can be used in a policy definition, when we need to retrieve information from EMOTIVE.

5.3.3.1 int size()

This function is used in order to retrieve the number of active nodes in the cloud.

```
List<URI> nodes = super.accounter.getNodesURI();
int maxTry = nodes.size();
return maxTry;
```

accounter variable is defined in the *Scheduler* class:

```
protected Accounting accounter;
```

It has been necessary to use the *getNodesURI()* method of the *Accounting* class because it gets information from the database in which data about the nodes are stored.

```
public List<URI> getNodesURI() {
        List<URI> nodes = new ArrayList<URI>();
        try {
            PreparedStatement nodesGet = connection.
               prepareStatement("SELECT uri FROM Nodes ORDER BY
               uri;");
            ResultSet rs = nodesGet.executeQuery();
            while(rs.next()) {
                nodes.add(new URI(rs.getString(1)));
            }
            return nodes;
        } catch (Exception ex) {
            Logger.getLogger(Accounting.class.getName()).log(
               Level.SEVERE, null, ex);
            return null;
        }
    }
```

5.3.3.2 URI getURI(int i)

Since nodes are stored in a *List* of URI it is possible to access elements by their integer index (position in the list). This function can be used to get the URI of a node using its index.

```
List<URI> nodes = super.accounter.getNodesURI();
URI nodeToSelect = nodes.get(i);
return nodeToSelect;
```

5.3.3.3 int numVMnode(int/URI i)

This function can be used when we need to know how many VMs are running in a node. We can have as a parameter the index of the URI List or directly the URI of the node we need.

This is the part of code that performs the main task, retrieving all the Domains (VMs) running in a node by using the Virtualization Manger client side:

```
VtMRESTClient vtm = new VtMRESTClient(node.getHost(), node.
    getPort());
Integer numVM = vtm.getDomains().size();
return numVM;
```

5.3.3.4 float freeCPU(int/URI i)

This function is needed to get the percentage of free CPU in a node. The value is computed by getting the percentage of total amount of CPU of a node and subtracting the percentage of the amount of CPU used by all the VMs stored and running in that node.

In order to get the total amount of CPU of a node, the Virtualization Manager (VtM) has been involved. We added the following method in the VtM Client interface (*VtMRESTClient*) through which we will get the required value.

```
public String getFreeCPUS() {
        String free;
        WebResource resource = client.resource(this.getAddress())
            .path("/freeCPUs/");
        free = resource.accept("text/plain").get(String.class);
        return free;
}
```

When this request is sended, the *VtM* (server side) handles it returning the percentage of CPU available in the node. It is simply computed by multiplying the number of CPUs in the node by 100 (Resource Manager does not estimate the maximum percentage over 100 but over 100 times number of CPUs).

```
    public int freeCPU() {
        int freeCPU = virt.getNodeCPUNum() * 100;
        return freeCPU;
    }
```

We can retrieve information about a VM using the *getMeasuredValues(vmid)* method. It returns an object *Cluster* which contains the list of VMs present in the node:

```
private List<Host> hosts;
```

This list contains information and data about VMs, stored in a *Metric* object. The following lines show the structure of a metric.

```
<HOST ID="vm7" IP="0.0.0.0" LOCALTIME="1343145719605" NAME="vm7">
<METRIC DMAX="0" MEAN="3000.00" NAME="cpu_speed" SOURCE="
   ResourceMonitor" TMAX="10190" TYPE="uint32" UNITS="MHz" VAL
   ="3000"/>
<METRIC DMAX="0" MEAN="4.00" NAME="cpu_num" SOURCE="
   ResourceMonitor" TMAX="10190" TYPE="uint16" UNITS="" VAL="4"/>
<METRIC DMAX="0" MEAN="" NAME="os_release" SOURCE="
   ResourceMonitor" TMAX="10190" TYPE="string" UNITS="" VAL
   ="2.6.20-xen3.1"/>
<METRIC DMAX="0" MEAN="256.00" NAME="mem_total" SOURCE="
   ResourceMonitor" TMAX="10190" TYPE="uint32" UNITS="KB" VAL
   ="256"/>
<METRIC DMAX="0" MEAN="23515.00" NAME="bytes_in" SOURCE="
   ResourceMonitor" TMAX="10190" TYPE="uint32" UNITS="bytes" VAL
   ="23515"/>
<METRIC DMAX="0" MEAN="0.00" NAME="bytes_out" SOURCE="
   ResourceMonitor" TMAX="10190" TYPE="uint32" UNITS="bytes" VAL
   ="0"/>
<METRIC DMAX="0" MEAN="300.00" NAME="disk_total" SOURCE="
   ResourceMonitor" TMAX="10190" TYPE="double" UNITS="GB" VAL
   ="300.00"/>
<METRIC DMAX="0" MEAN="99.51" NAME="mem_used" SOURCE="
   ResourceMonitor" TMAX="10190" TYPE="float" UNITS="%" VAL
   ="99.51"/>
<METRIC DMAX="0" MEAN="" NAME="vm_state" SOURCE="ResourceMonitor"
    TMAX="10190" TYPE="string" UNITS="" VAL="Running_Unavailable
   "/>
<METRIC DMAX="0" MEAN="0.04" NAME="cpu_user" SOURCE="
   ResourceMonitor" TMAX="10190" TYPE="float" UNITS="%" VAL
   ="0.04"/>
<METRIC DMAX="0" MEAN="4.00" NAME="cpu_vnum" SOURCE="
   ResourceMonitor" TMAX="10190" TYPE="uint16" UNITS="" VAL="4"/>
<METRIC DMAX="0" MEAN="" NAME="machine_type" SOURCE="
   ResourceMonitor" TMAX="10190" TYPE="string" UNITS="" VAL="
```

```
   x86_64"/>
<METRIC DMAX="0" MEAN="" NAME="os_name" SOURCE="ResourceMonitor"
   TMAX="10190" TYPE="string" UNITS="KB" VAL="linux"/>
<METRIC DMAX="0" MEAN="0.00" NAME="cpu_assigned" SOURCE="
   ResourceMonitor" TMAX="10190" TYPE="float" UNITS="%" VAL
   ="0.00"/>
<METRIC DMAX="0" MEAN="1024.00" NAME="swap_total" SOURCE="
   ResourceMonitor" TMAX="10190" TYPE="uint32" UNITS="KB" VAL
   ="1024"/>
</HOST>
```

In this function the relevant value is the "cpu_user" which contain the amount of cpu used by a VM. We can now see in details the structure of the method through which we get the percentage of available CPU in a node.

```
// get total available CPU of the node
VtMRESTClient vtm = new VtMRESTClient(node.getHost(),
    node.getPort());
Integer cpu = Integer.parseInt(vtm.getFreeCPUS());
Float cpu_free = Float.parseFloat(cpu.toString());
// get measured data for the node

List<OVFWrapper> domains = vtm.getDomains();

for (OVFWrapper o : domains) {
    String vmid = o.getId();
    String xml = vtm.getMeasuredValues(vmid);
    Cluster c = RMXMLConversor.parseXML(xml);
    List<Host> hosts = c.getHosts();

    // for each VM in the node, read used cpu by that
        Vm
    // and make the difference with the total Cpu of
       the node.
    for (Host h : hosts) {
        if (h.getMachineId().equals(vmid)) {
            h.getMetrics();
            List<Metric> met = h.getMetrics();
            for (Metric m : met) {
                if (m.getName().equalsIgnoreCase("
                   cpu_user")) {
```

```
                                cpu_free -= Float.parseFloat(m.
                                    getValue().toString());
                            }
                        }
                    }
                }
            }
            return cpu_free;
        }
```

5.3.3.5 void destroyVM(string vmId)

This function can be used to destroy a VM. It simply uses the *destroy* function of the default scheduler passing as parameters the id of the VM to be destroyed and *userDN* which is a security session information about the user:

```
super.destroy(vmId, userDN);
```

5.3.3.6 boolean exists(string id)

Using this function is possible to check if a node or a VM is present in the network. The parameter is a string that can represent the URI of a node or a VM identifier. If we need to check for a node this function just accesses to the list of nodes and check if the one we need is present.

```
List<String> nodes = super.getNodes();
for (String n : nodes) {
        if (id.equalsIgnoreCase(n))
                return true;
        }
```

Contrariwise if we are looking for a VM, it tries to get the node in which the VM, whose id has been passed to the *exixsts* function, is running.

```
URI node = getNodeVM(id, userDN);
return true;
```

If the VM is not found an exception is thrown and it returns *false*. Listing 5.5 shows how the *URI getNodeVM(string id, string userDN)* has been implemented.

Listing 5.5: Method getNodeVM()

```
protected URI getNodeVM(String vmId, String userDN) throws
   VRMMSchedulerException {
        URI nodeURI = accounter.getVMNode(vmId, userDN);

        if (nodeURI != null) {
            return nodeURI;
        }
        //if machine is not found in the database, looks for it
           in the VtMs
        for (URI node : accounter.getNodesURI()) {
            try {
                VtMRESTClient vtm = new VtMRESTClient(node.
                   getHost(), node.getPort());
                OVFWrapper ovf = vtm.getDomain(vmId);
                if (ovf != null) {
                    accounter.addVirtualMachine(node, new
                       EmotiveOVF(ovf));
                    return node;
                }
                vtm.destroy(vmId);
            } catch (Exception e) {
                Logger.getLogger(Scheduler.class.getName()).log(
                   Level.WARNING, "Can't connect with " + node +
                   " : " + e.getMessage());
            }
        }

        accounter.removeVM(vmId,userDN);
        throw new VRMMSchedulerException("VM \"" + vmId + "\"
           cannot be found in any node.");
    }
```

5.3.3.7 int numVMs()

In order to get the number of VMs in the system this function can be used. It just exploit the function *getAllDomains()* developed in the *SimpleScheduler* class which returns the domains (VMs) in the EMOTIVE cloud. By counting them we get the total number of VMs.

5.3.3.8 string getVMId(int i)

This function gives the possibility to obtain the id of a virtual machine using the same process of the function *URI get(int i)*. It is just necessary to get the id by accessing to the list of domains using the index passed as a parameter:

```
List <OVFWrapper> list = super.getAllDomains();
String id = list.get(i).getId();
```

5.3.3.9 float getVMCPU(int/string i)

In order to retrieve the cpu usage of the VM we need to get the *Metric* object in which informations about VMs in a node are stored. For this reason we need to know the node in which the VM is and read its information as we have seen in the function which allow us to get the free CPU of a node (5.3.3.4).

```
URI node = getNodeVM(id, userDN);
VtMRESTClient vtm = new VtMRESTClient(node.getHost(),
    node.getPort());

String xml = vtm.getMeasuredValues(id);
Cluster c = RMXMLConversor.parseXML(xml);
List<Host> hosts = c.getHosts();
for (Host h : hosts) {
    if (h.getMachineId().equals(id)) {
        h.getMetrics();
        List<Metric> met = h.getMetrics();
        for (Metric m : met) {
            if (m.getName().equalsIgnoreCase("
                cpu_user")) {
                return Float.parseFloat(m.getValue().
                    toString());
            }
        }
    }
}
```

Using this function we can give as a parameter a *string* representing the id of the VM or an *int* which is the index through which we can access to the list of all the domains. This is also valid for the next method (int getVMMem(int/string id)).

5.3.3.10 int getVMMem(int/string id)

The *int* value returned by this function is the amount of memory used by a VM whose id or index is *id*. It is computed reading this information stored in the *OVFWrapper* object of the VM and to which we can access by the method *int getMemoryMB()*.

```
// get the OVFWrapper of the VM
OVFWrapper o = getDomain(id, userDN);
//get VM memory
return o.getMemoryMB();
```

5.3.3.11 int freeMem(int/URI i)

Besides free CPU in a node, information about the free memory can be useful in order to write a scheduling policy. By the function *freeMem* we can obtain the amount of CPU in a node that is free and can be still used (E.g. storing other VM). As the previous functions we can have a *URI* or a *int* value as a parameter which respectively represent the URI of the interested node and its index in the list of nodes. It has been necessary to add some code in the *VtM* module in order to retrieve this information. The core of this function is shown in the next listing in which the free memory in a node in computed by subtracting memory used by the VMs running in that node to the total memory that the node support.

```
public int freeMemory() {
        int freeMemory = virt.getNodeMemory();
        for (EmotiveOVF ovfDomain : getDomains().
            getRunningDomains()) {
             freeMemory -= ovfDomain.getMemoryMB();
        }
        return freeMemory;
    }
```

6

Case studies to test the policy framework

In this section we will show how the new policy framework works, considering different policy definitions that have been using in order to test the implemented features. In order to deal with realistic operations we took inspiration from scheduling operation used in other platforms, such as the ones described in section 2.2. Tests have been done in a cluster consisting of 2 servers: *pctomeu* and *pctinet* as shown in figure 6.1.

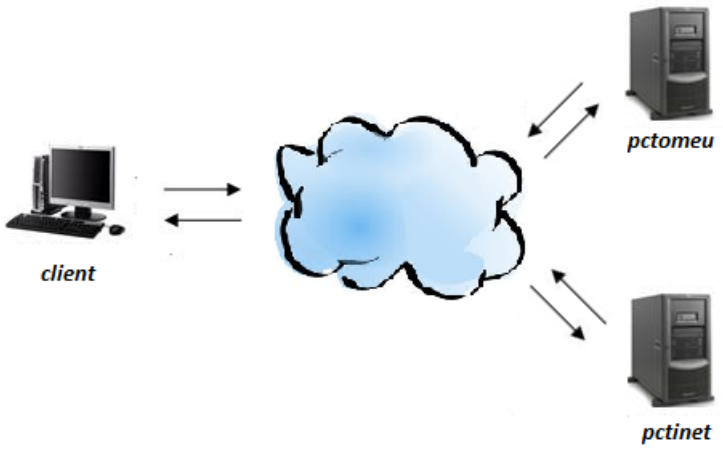


Figure 6.1: Cluster for testing

Hardware and software characteristics of the servers are described in table 6.1.

	pctomeu	pctinet
Kernel	Linux 2.6.18-xen	Linux 2.6.18.8-xen
Processor	Intel(R) Xeon(R) CPU X5355	Intel(R) Xeon(R) CPU 5160
Clock speed	2.66 GHz	3.00 GHz
Number of processors	4	4
RAM	12820 MB	14618 MB
Libvirt library	libvir 0.7.6	libvir 0.7.7
Hypervisor	Xen 3.1.0	Xen 3.3.0

Table 6.1: Servers specifications

6.1 Packing Policy

Using a packing policy, new VMs will be created in the node with more VMs running. This can be useful to reduce the number of active nodes in the system. The definition of this policy, using LEPIC language is shown in listing 6.1.

Listing 6.1: Packing Policy

```
/* Packing -> Use node with more VMs running first */

policy policy_packing;
type simple(operation create){

  URI nodeToSelect;
  int i,node,num;
  int size;
  int min;

      min = 0;
      size = size();

      for (i=0; i < size; i++)
        {
        num = numVMnode(i);

        if (num > min)
            {
              min = num;
              node = i;
```

```
            }
        }
    nodeToSelect = getURI(node);
    return nodeToSelect;

}
```

This has been tested setting up a system with 2 VMs created in *pctinet*. Then, loaded the policy, we create a new VM (*vm3*), which, according to the scheduling operations it will be store in *pctinet* because it has more VMs running than *pctomeu*, as we can see in figure 6.2.

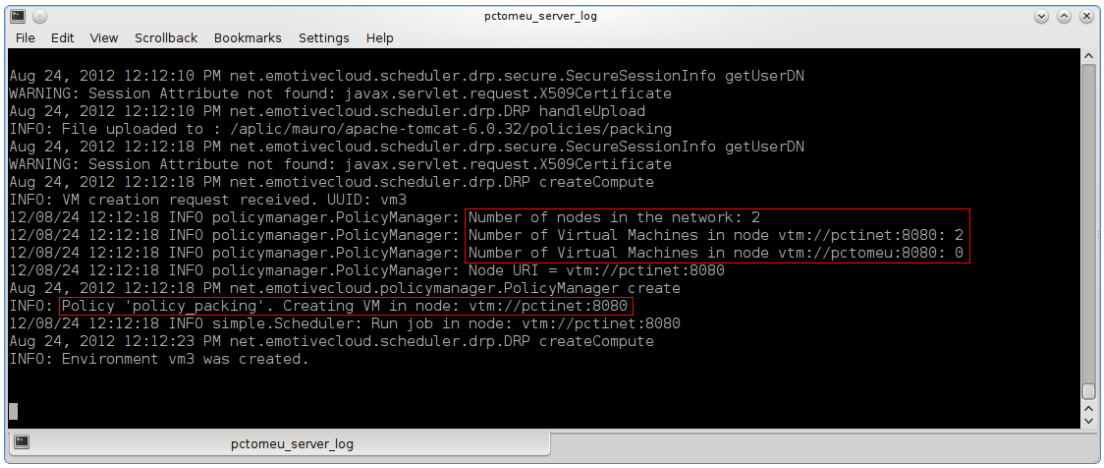


Figure 6.2: Server log during the creation of vm3

We can see in snapshot 6.3 that *vm3* has been created and it is running in *pctinet* as expected.

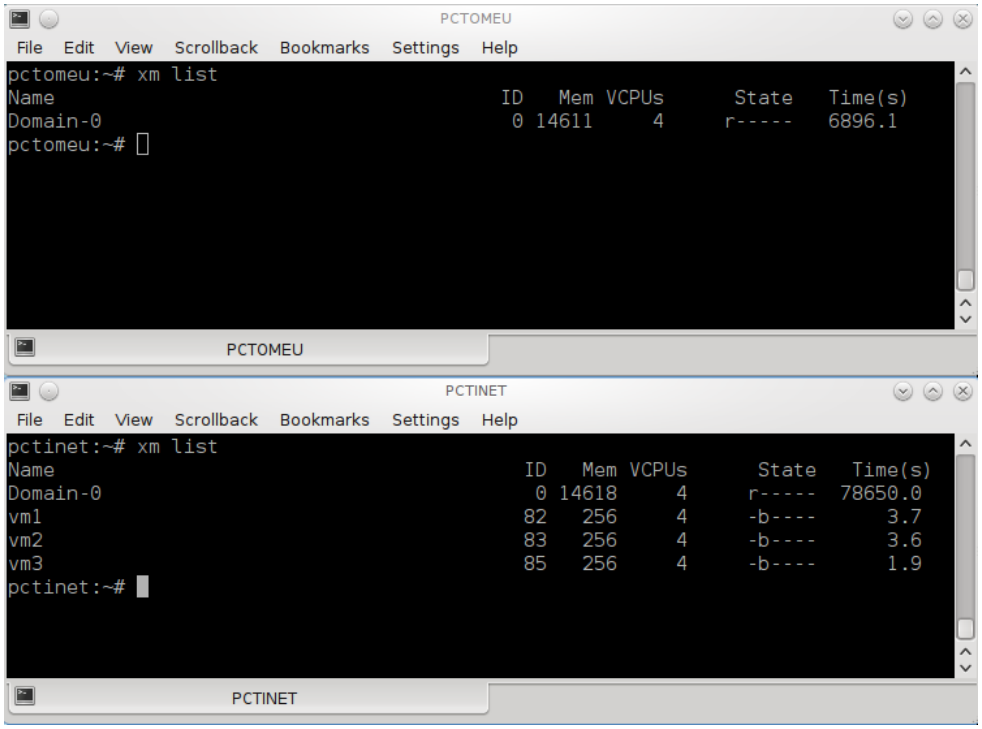


Figure 6.3: VMs placement using packing policy

6.2 Striping Policy

In order to maximize the resources available to VMs in a node, the striping policy (defined in listing 6.2) can be convenient because nodes with less VMs running are used first.

Listing 6.2: Striping Policy

```
/* Striping -> Use node with less VMs running first */

policy policy_striping;
type simple(operation create){

  URI nodeToSelect;
  int i,node,num;
  int size;
  int max;

      max = 100;
```

```
      size = size();

      for (i=0; i < size; i++)
        {
        num = numVMnode(i);

        if (num < max)
            {
              max = num;
              node = i;
            }
        }
    nodeToSelect = getURI(node);
    return nodeToSelect;
  }
```

Before testing this scheduling policy we created 2 VMs in *pctinet* as we can see in sceenshot 6.4

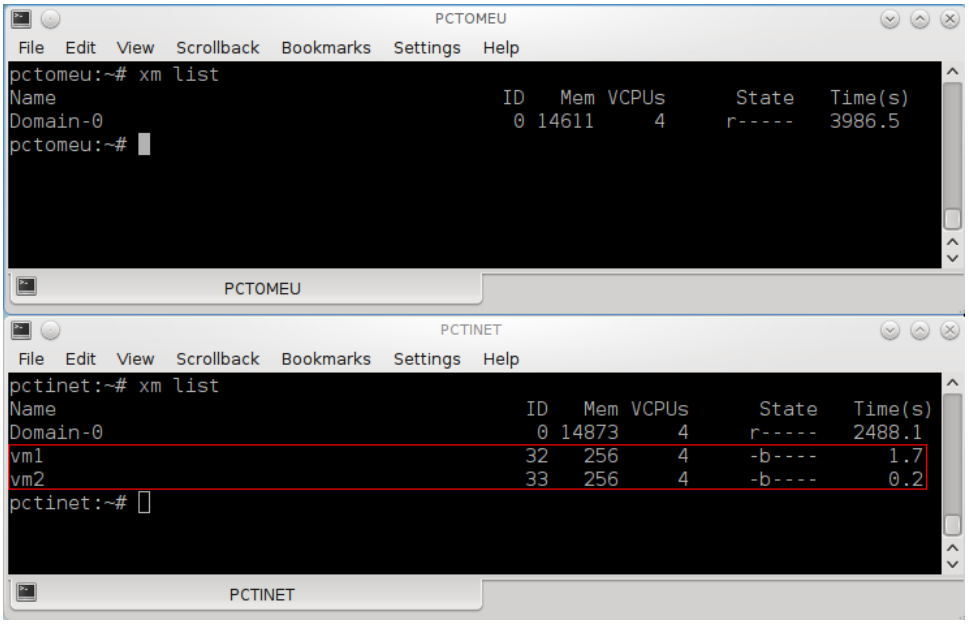


Figure 6.4: VMs placement before striping policy

Then we create 2 more VMs and following the policy strategy they will be created in *pctomeu* which has less VMs running (0 in this case). Snapshot 6.5 show the server log during the creation of the fourth VM (vm4).

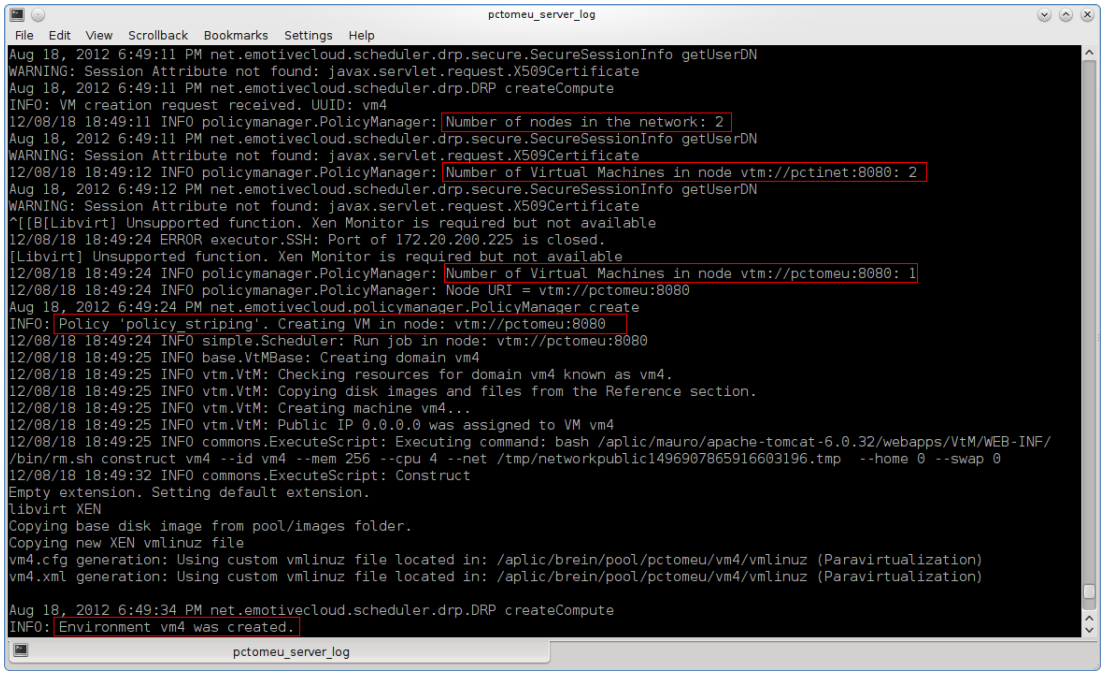


Figure 6.5: Server log during the creation of vm4

Therefore we will have 2 VMs in each host as we can see in image 6.6.

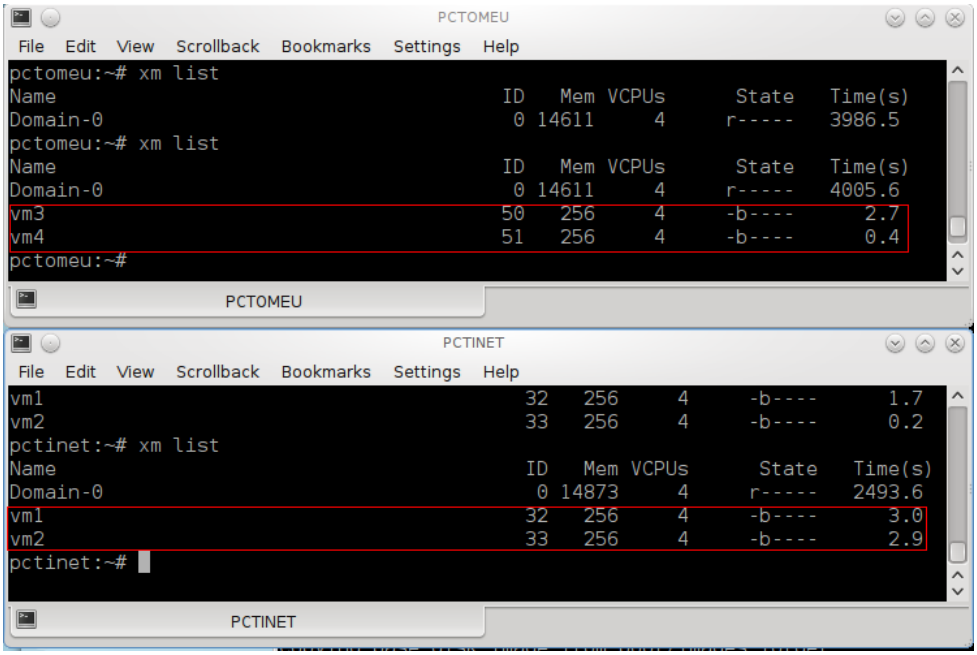


Figure 6.6: VMs placement using striping policy

6.3 Load-aware Policy

This policy has been created considering the load of a node and it chooses to create VMs in the node with more free CPU first.

Listing 6.3: Load-aware Policy

```
 /* CPU -> Use node with less CPU Load running first */

policy policy_CPU;

type simple(operation create){

  URI nodeToSelect;
  int i,node;
  int size;
  float min;
  float num;

      size = size();
      for (i=0; i < size; i++)
        {
        num = freeCPU(i);

        if (num > min)
            {
              min = num;
              node = i;
            }
        }
    nodeToSelect = getURI(node);
    return nodeToSelect;
  }
```

In order to test this policy we created 1 VM in each server (*vm1* in *pctinet*, *vm2* in *pctomeu*) as we can see in snapshot 6.7.

Then we stressed the CPU load of *vm2* by using the command *dd if=/dev/zero of=/dev/null*: image 6.8.

Then, when we try to create a new VM, the node with more free CPU will be chosen. In this case, as we can see in screenshot 6.9, we have free CPU values equals to 399.97 for *pctinet* and 185.12 for *pctomeu* so the next eligible host will be *pctinet*.

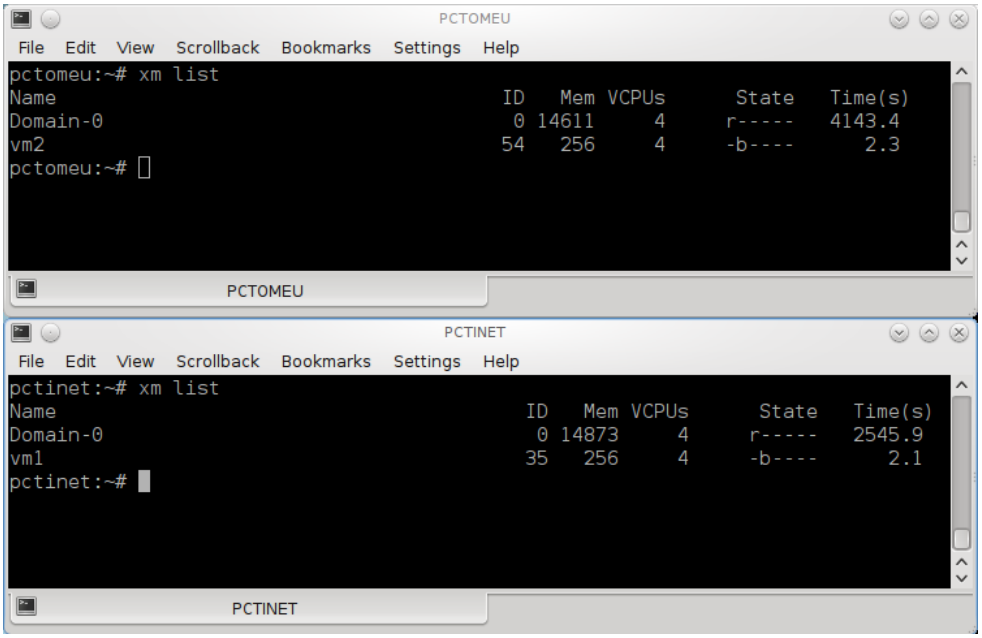


Figure 6.7: VMs placement before Load-aware policy

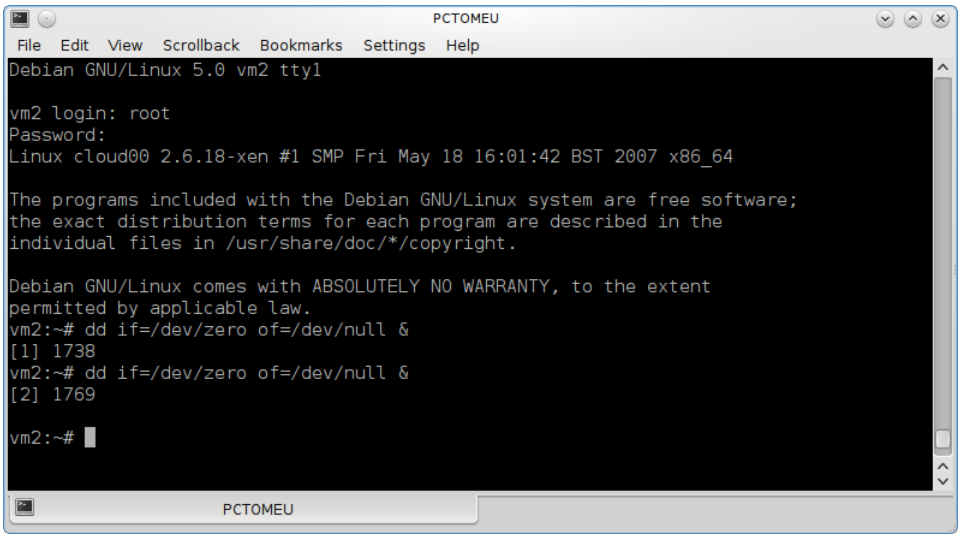


Figure 6.8: Stressing vm2

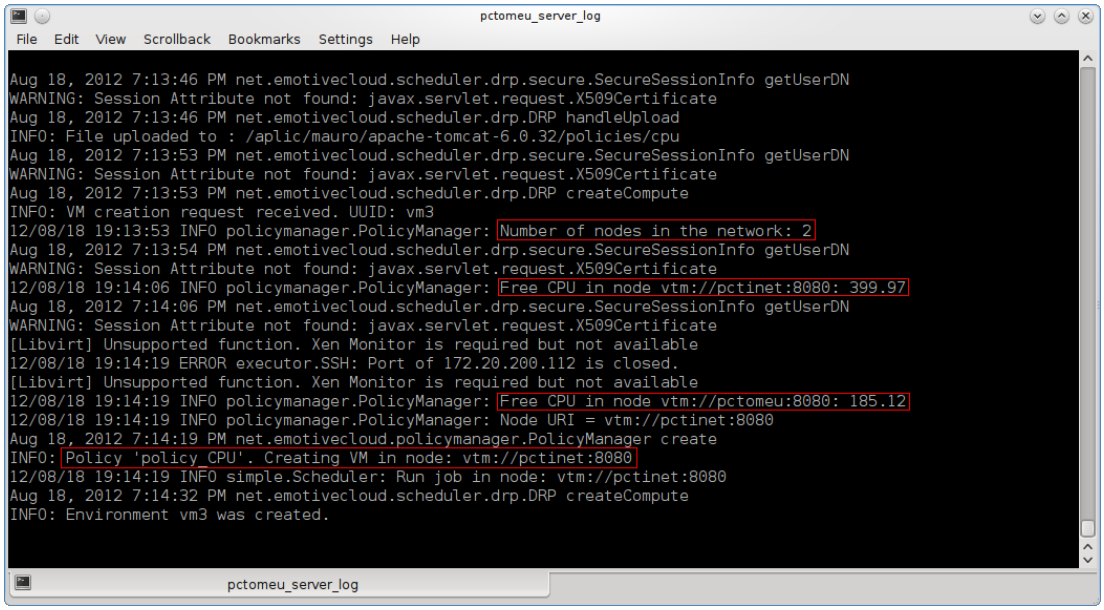


Figure 6.9: VMs placement running Load-aware policy

6.4 Destroy policy

Unlike the previous policies which deal with creation of VMs, this one performs an operation *destroy*. When this is loaded in the system and the *destroy* command is typed in the console (image 6.10), the VM with the highest usage of CPU will be destroyed.

Listing 6.4: Destroy Policy

```
/* Destroy VM with highest CPU usage */

policy destroy_policy;
type simple(operation destroy){

int i, n, index;
float cpu, min;
string d;

  index = 0;
  min = 0.0;
  n = numVMs();
  for (i=0; i < n; i++){
    cpu = getVMCPU(i);
```

```
    if (cpu > min){

      index = i;
      min = cpu;
    }
  }

  d = getVMId(index);

return d;
}
```

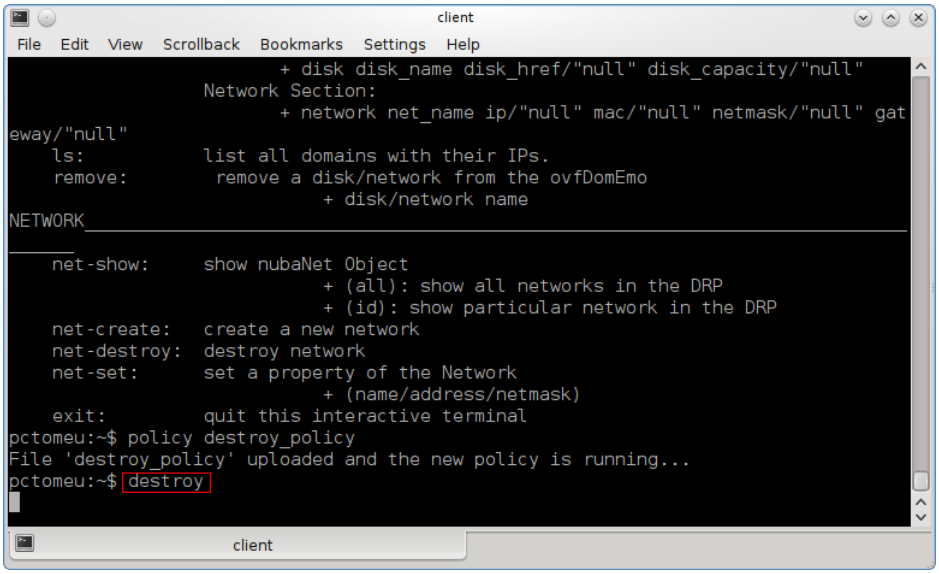


Figure 6.10: New destroy command in EMOTIVE

In order to test this new feature we created 3 VMs (*vm1*, *vm2* and *vm3*) and we stressed the CPU of *vm2* as we did in the previous policy. Then when the policy runs, firstly it gets the number of VMs in the system, secondly it computes the percentage of free CPU for all of them, destroying the one with highest value (in this case *vm2*; screenshot 6.11).

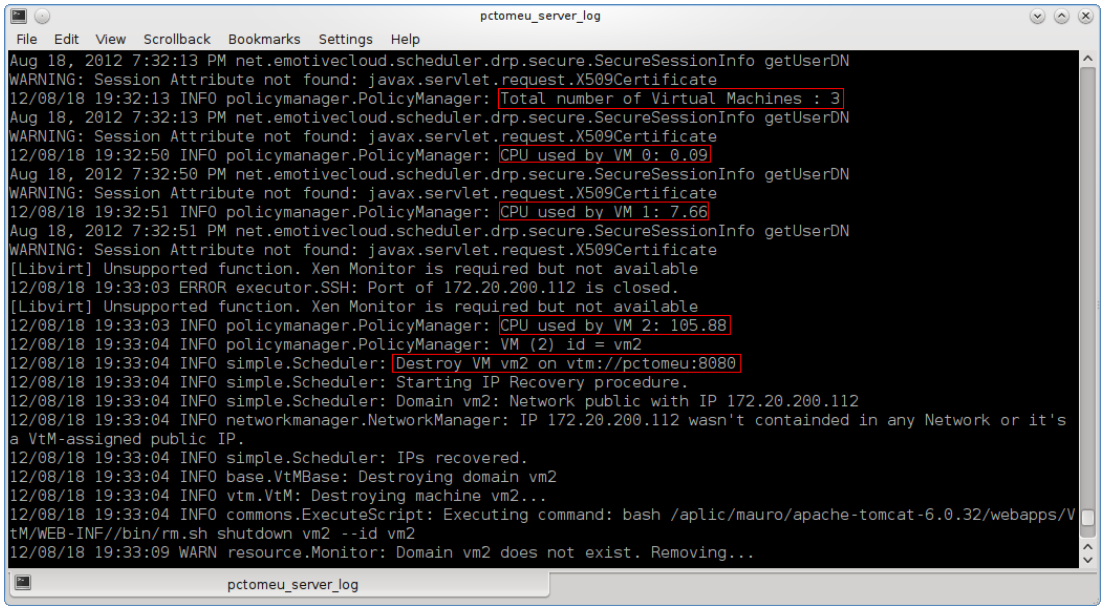


Figure 6.11: Server log for destroying operation

6.5 Monitor Policy

In order to demonstrate how a monitoring scheduling operation works, we defined a set of operations that not allow the existence of VMs which need more than 512 MB of memory. In order to do that this policy checks every *5000* ms the memory required by every VMs present in the system. If this value is greater than 512, the corresponding VM is destroyed.

Listing 6.5: Monitoring Policy

```
policy monitoring_policy;
type monitor CheckMem (time 5000){

int i,n,mem;
string vm;

  n = numVMs();
  for (i=0; i < n; i++){

    mem = getVMMem(i);

    if (mem > 512){
```

```
        vm = getVMId(i);
        destroyVM(vm);
      }
    }
}
```

As we can see in the snapshot 6.12, when we create a VM (*vm1*) which required 640 MB of memory it will be destroyed as soon the monitoring policy find it out (in this case the worst case is after 5000 ms) and it will not be present in the system.

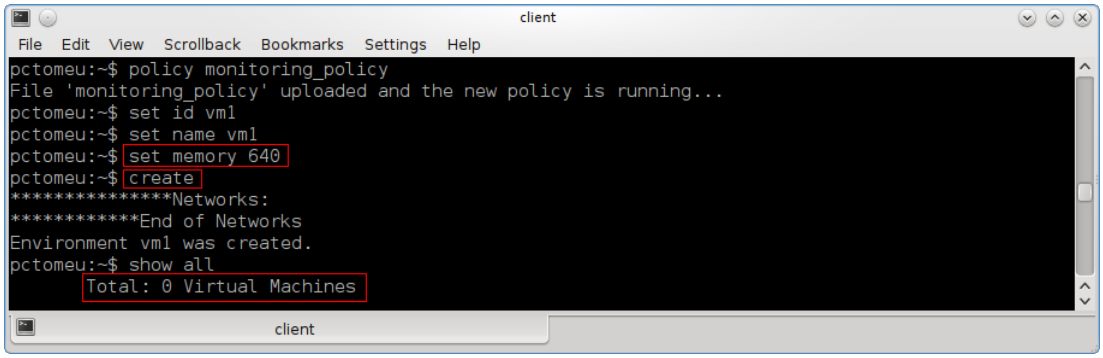


Figure 6.12: Monitoring policy example

More in detail, taking a look to the server log 6.13, we can see how operations are performed. At the beginning no VMs are present in the system and the function *ChechMem* of the monitoring policy is running. Then when the creation of *vm1* is detected by that monitoring function it is destroyed.

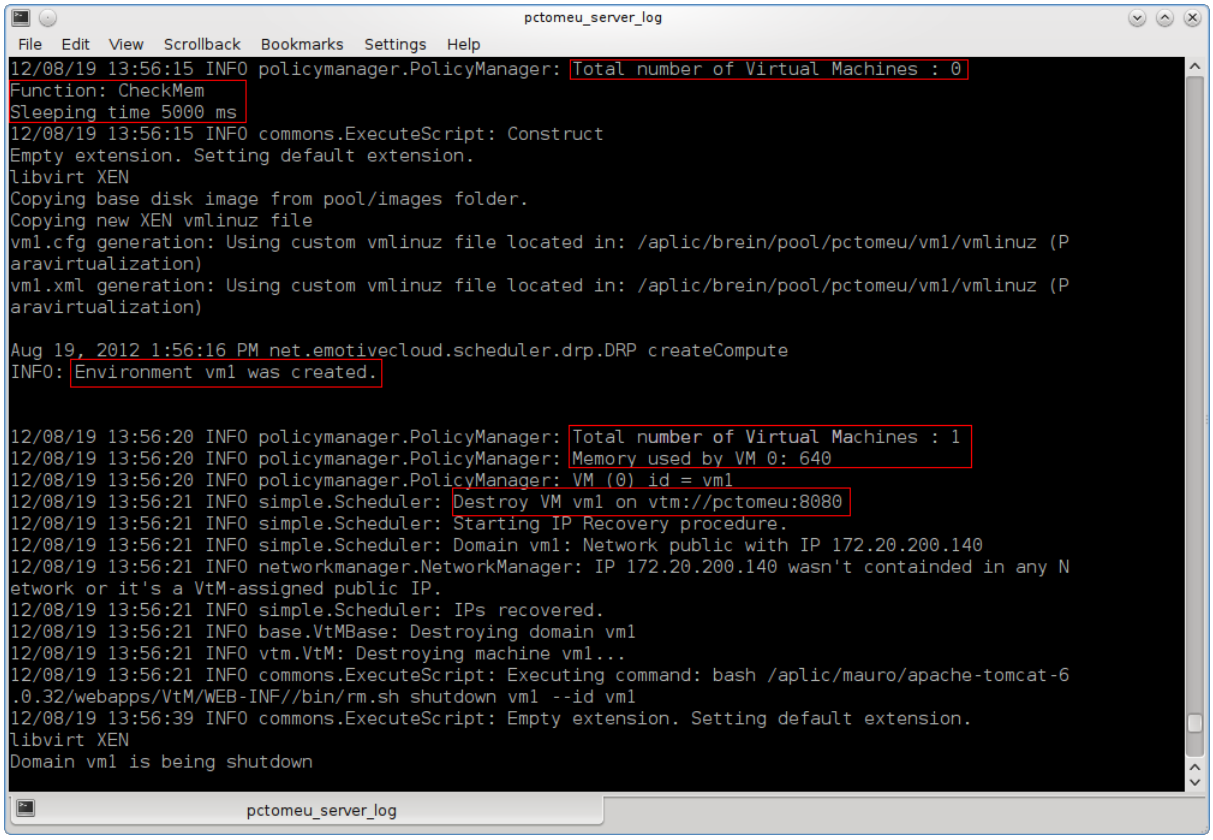


Figure 6.13: Server log running monitoring policy

6.6 Extended policy

This policy handles both simple and monitoring parts performing a Round Robin algorithm for creation of VM and checking every *10000* ms percentage of CPU used by all the VMs. If it is more than 50.00, corresponding VM will be destroyed.

Listing 6.6: Destroy Policy

```
policy extended_policy;

type simple(operation create){

/* Create VMs using a Round Robin algorithm */

URI nodeToSelect;
int lastSelected=0;
```

```
int size;


    size = size();
    nodeToSelect = getURI(lastSelected % size);
    lastSelected = lastSelected +1;

    return nodeToSelect;

}

type monitor Nodes (time 10000){

/* Destroy a vm when its cpu load it higher than 50 */

int i,n;
string vm;
float cpu;
  n = numVMs();
  for (i=0; i < n; i++){

    cpu = getVMCPU(i);

    if (cpu > 50.00){
      vm = getVMId(i);
      destroyVM(vm);
    }
  }

}
```

As we can see in snapshot 6.14 Round Robin algorithm simply reads the last node used to create a VM and chooses the next one. In the meanwhile function *Nodes* of the monitoring part keeps running.

In order to test the monitoring part we just stress the CPU of *vm1* as in the previous cases and the policy will detect the new state (snapshot 6.15).

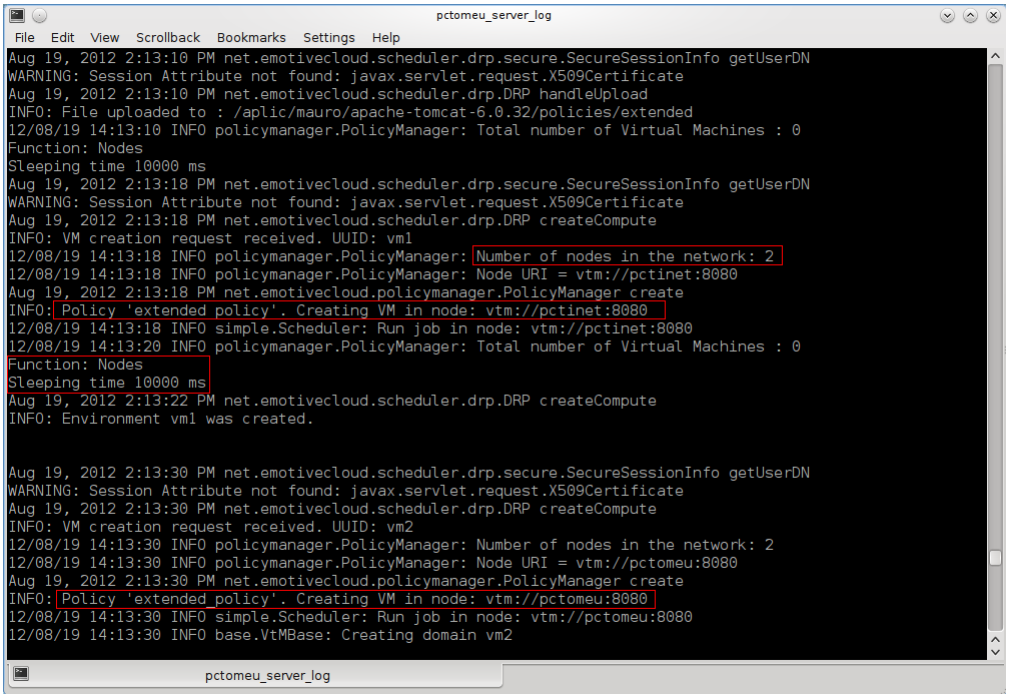


Figure 6.14: Server log 1 running extended policy

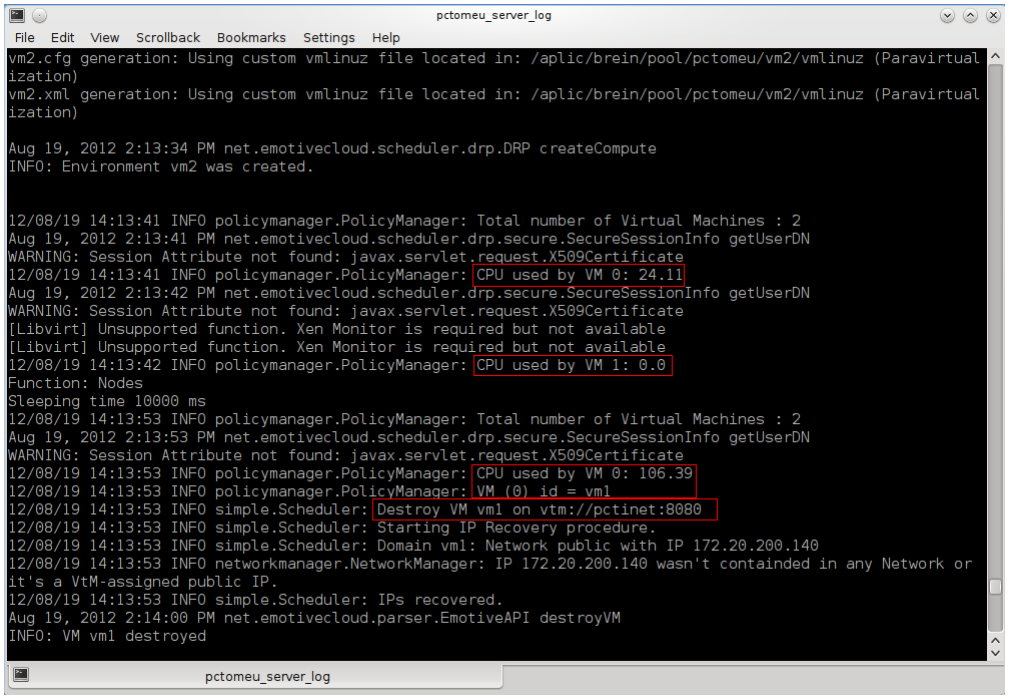


Figure 6.15: Server log 2 running extended policy

7

Analysis and performance tests

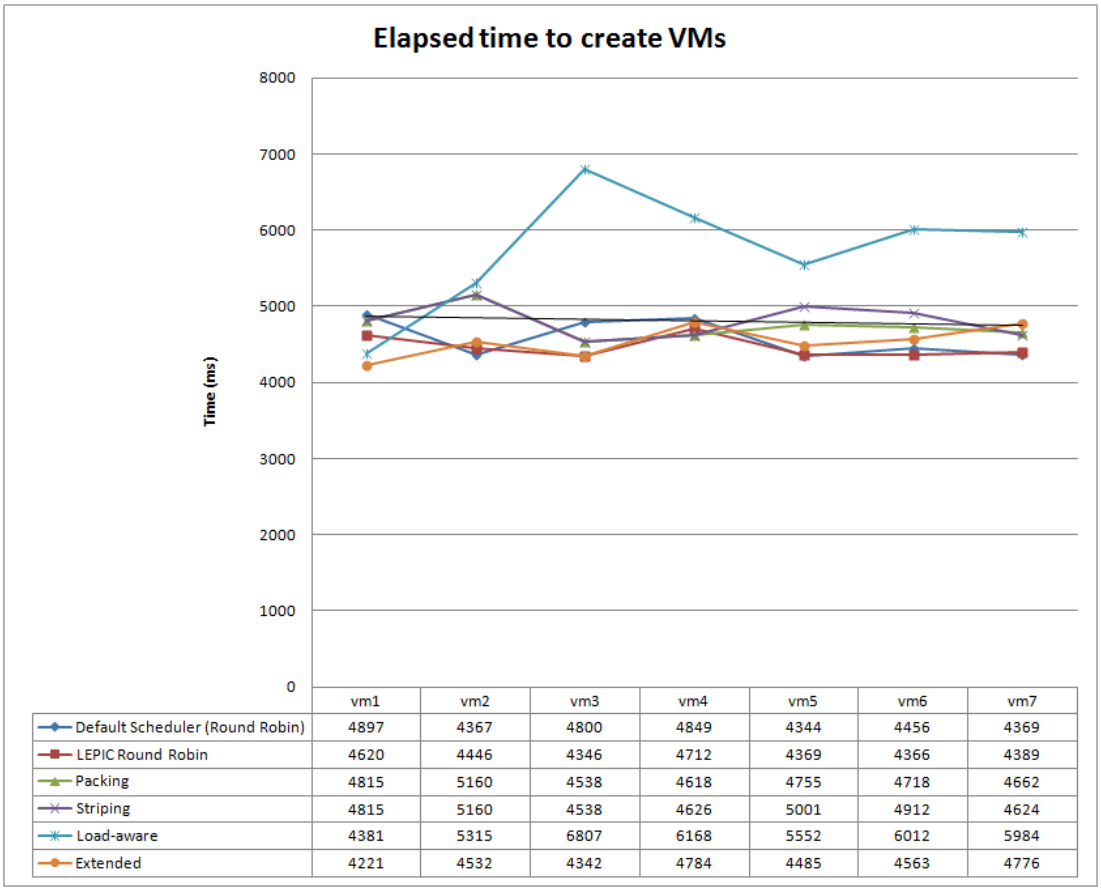
In order to evaluate the overhead introduced by the framework, some parameters have been tested using the different policies showed in previous chapter. First of all we noticed that the creation of a VM using the developed policy engine is not affected by delays and we have the same performance as the original EMOTIVE scheduling layer.

As we can see in graph 7.1, that shows the time spent to create a VM, the trend of the default scheduler, implementing a simple Round Robin algorithm, and the Round Robin policy defined using LEPIC language is similar and overhead is negligible. However we also have a similar trend for *packing* and *striping* policies while, since *load-aware* policy needs to get information from the system, such as free CPU in a node, higher times to create a VM using that policy are expected. About the *extended* policy we can notice that the monitoring part does not cause delays when we create a VM.

7.1 Policies comparison

This benchmark compares the CPU usage, which is related to power consumption, of servers involved in the test in order to check the overload due to the developed framework and to see which policy is preferable to our needs. In the comparison we used a workload that creates 4 VM(s) using first *packing* policy and then the *striping* one. Within each virtual machine runs a job which places a configurable load on different parts of the system, such as CPU and disk, stressing them.

If we want to minimize the number of cluster nodes in use, a *packing* policy may be used. As a matter of fact, as we can see in graph 7.2, using that scheduling approach



	vm1	vm2	vm3	vm4	vm5	vm6	vm7
Default Scheduler (Round Robin)	4897	4367	4800	4849	4344	4456	4369
LEPIC Round Robin	4620	4446	4346	4712	4369	4366	4389
Packing	4815	5160	4538	4618	4755	4718	4662
Striping	4815	5160	4538	4626	5001	4912	4624
Load-aware	4381	5315	6807	6168	5552	6012	5984
Extended	4221	4532	4342	4784	4485	4563	4776

Figure 7.1: Elapsed time to create VMs

all the VMs have been created in *pctinet* and they are using its resources and CPUs. Contrariwise, since no VMs are running, CPUs load of *pctomeu* is very low and in order to save power this node may be turned off.

If we want to maximize the resources available to VM we may use the *striping* policy which spreads the VMs in the cluster nodes. As we can see in graphs 7.3, the load level of CPUs in each server is fairly the same. We can notice that CPUs load in *pctomeu* is a bit higher than in *pctinet*. This is reasonable and it is due to the different architecture of the servers, such as different processors (clock speed of 2.66 GHz in *pctomeu* and of 3.00 GHz in *pctinet*).

In order to evaluate performance and overhead due to monitoring operations defined using LEPIC language and loaded in EMOTIVE, we tested the system using the *extended* policy described in section 6.6.

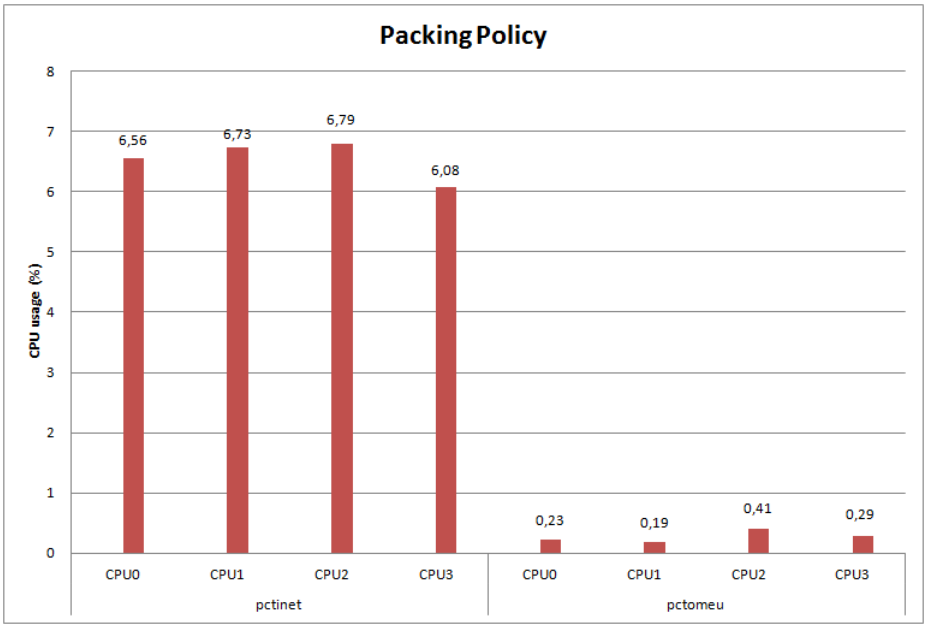


Figure 7.2: CPU load using packing policy

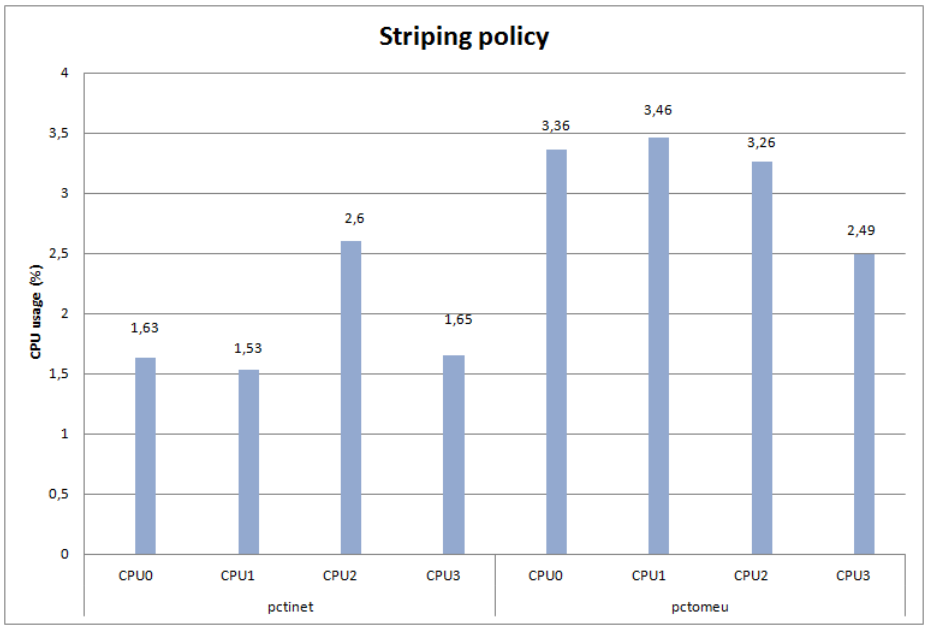


Figure 7.3: CPU load using striping policy

It checks every 10 seconds if the CPU load of a VM is higher than 50%. If it is, that VM will be destroyed. The benchmark started running the default scheduler and 2 VMs

have been created in *pctomeu* (*vm2* and *vm4*) and 2 VMs in *pctinet* (*vm1*, *vm3*). Then, after 5 seconds, we loaded in EMOTIVE the *extended* policy which started to check CPUs usage of every VMs. We can notice it in figure 7.4, in which every 10 seconds we have peaks in the CPU load trend due to the monitoring operation.

At second 47 we stressed the CPU of *vm2*. This was detected by the scheduler which, according with the policy, destroyed it. We can seen it in the graph, in which, between second 47 and 57, we have higher usage of CPU in *pctomeu* due precisely to the destroying operation. After that the behaviour of the system continues to be the same as before.

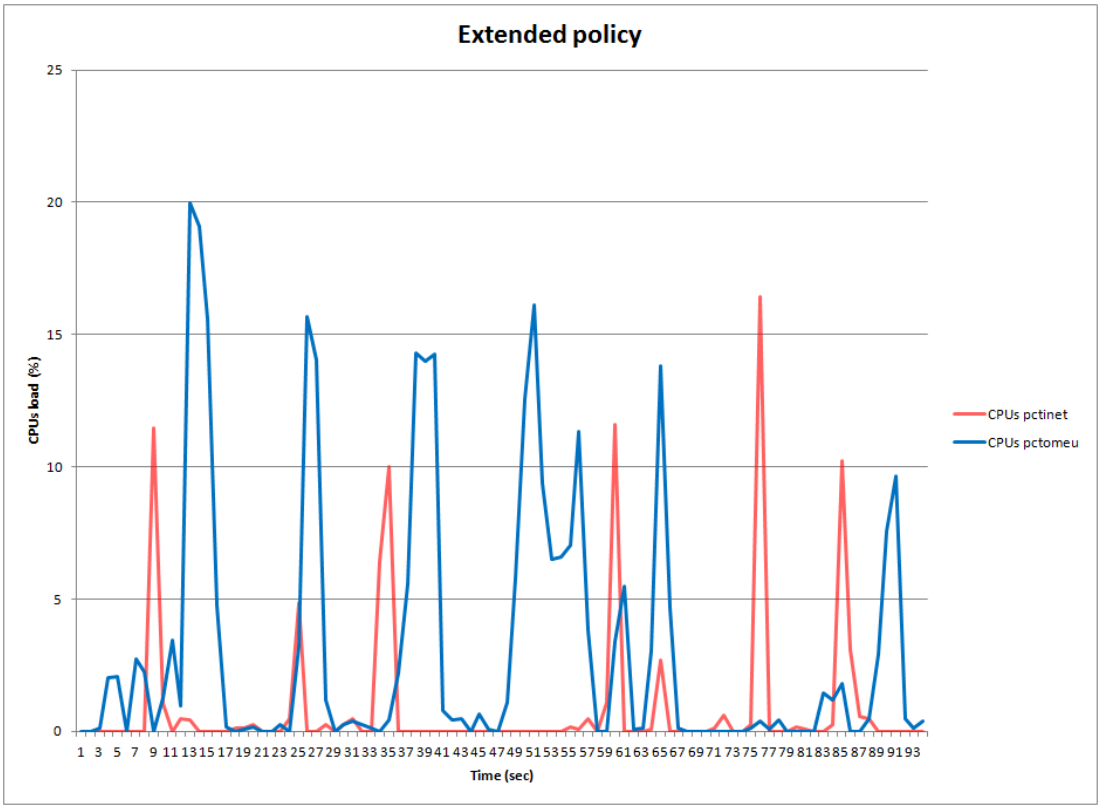


Figure 7.4: CPU load running extended policy

Although this benchmark it is based in a specific case where each VM does the same kind of job and do not mind in what servers is running, it was useful to understand the overhead introduced by the new framework, especially using the *extended* policy, and to evaluate the load of each server depending on the adopted scheduling policy. Moreover we should also consider the space overhead due to policy storage in the server

because, as we explained, when a new policy defined with LEPIC language is loaded in EMOTIVE it is stored in a directory. We decided to maintain this approach because it can be useful to keep all the policy definitions either because it could be useful to analyze them in case of failure or scheduling problems or because we might want to create a policy similar to an existing one just by editing it, without the need to redefine it from scratch.

8

Conclusions

8.1 Summary

The future of cloud computing is moving toward more ubiquity, as greater demands from customers and greater capabilities from providers unfold.

Thanks to this project new features have been implemented in the EMOTIVE middleware carrying out research for new functionalities that can provide users with new capabilities that have not been deployed in other Cloud Computing frameworks. Since the research conducted by UPC and BSC also focuses on green computing, developing an architecture that can improve power consumption and performance at the same time, the policy engine that have been implemented allow us to load and enforce dynamically policies for virtual machines placement and scheduling in the Cloud infrastructure, giving us the possibility to change the system behaviour depending on our needs that can be related to network or nodes load as performance or power saving.

In a field such as Cloud Computing that evolves more and more these new features can be really profitable, especially since IaaS solutions we analyzed before starting this project do not support scheduling and policy definition but allow users only to use and exploit pre-defined operations. Moreover the possibility to create and load policy that can monitor parameters in the cloud, taking decisions depending on the system behaviour, gives EMOTIVE something more advanced than other frameworks.

8.2 Future work

Since EMOTIVE is very easy to extend thanks to its modular Web Service architecture and it is being used by BSC to do research in Cloud Computing, as well as in some research projects such as VENUS-C (VENUS-C, 2010-2012), OPTIMIS (OPTIMIS, 2010-2013), and NUBA (NUBA, 2009-2012), some features still need to be developed so that the framework developed in this project can be fully integrated exploiting all its functionalities.

Whereas this work provides EMOTIVE new features, it could be improved, especially taking into account performances and the validation part of the policy engine. Although the system has been tested defining some policies used in other real Cloud middleware, it would be desirable to define more complex policies and testing them. For instance it would be useful to define policies that exploit VMs migration (still not totally working in EMOTIVE) and power saving, turning on and off nodes depending on the needs.

Moreover if we want to be completely interoperable with EMOTIVE scheduling operations, we will need to extend LEPIC language implementing new functions that can be useful in policy definitions. This can be done by a mapping between EMOTIVE API and LEPIC function, improving the flexibility of the system.

References

[1] EMOTIVE Cloud, *http://www.emotivecloud.net/*, The BSC's IaaS open-source solution for Cloud Computing.

[2] XEN hypervisor, *http://www.xen.org/*.

[3] libvirt: The virtualization API, http://libvirt.org/.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Matei Zaharia, *Above the Clouds: A Berkeley View of Cloud Computing* , University of California, Berkeley: Technical Report No. UCB/EECS-2009-28, 2009.

[5] B. Sotomayor, R. S. Montero, I. M. Llorente, I. Foster, *Virtual Infrastructure Management in Private and Hybrid Clouds*,IEEE Internet Computing, vol. 13, no. 5, pp. 14-22, Sep./Oct. 2009. DOI: 10.1109/MIC.2009.119.

[6] EMOTVE Guide, *http://autonomic.ac.upc.edu/emotive/wp-content/uploads/2009/03/emotive.pdf*.

[7] OpenNebula, *http://opennebula.org/*, Project - OpenNebula: The Open Source Solution for Data Center Virtualization.

[8] OpenStack, *http://www.openstack.org/*, Open source software for building private and public clouds.

[9] Amazon EC2, *http://aws.amazon.com/ec2*, Amazon Elastic Compute Cloud

[10] Eucalyptus cloud computing software, *http://www.eucalyptus.com/*

[11] SableCC, *http://sablecc.org/*

[12] Alexandre VaquÃľ Brull, *The Green Evolution of EMOTIVE Cloud.*

[13] Virsh - management user interface, *http://linux.die.net/man/1/virsh.*

[14] Nicodemos Damianou, Naranker Dulay, Emil Lupu, Morris Sloman, *Ponder: A Language for Specifying Security and Management Policies for Distributed Systems.*

[15] OpenVPN *http://openvpn.net)*, OpenVPN Technologies, Inc. All Rights Reserved., 2002-2011.

[16] OCCI API, *http://occi-wg.org/*, Open Cloud Computing Interface.